Equilibrage de charge et redistribution de données sur plates-formes hétérogènes

Hélène Renard

N° d'ordre : 344
N° attribué par la bibliothèque : 05ENSL0 344

ÉCOLE NORMALE SUPÉRIEURE DE LYON
Laboratoire de l'Informatique du Parallélisme

THÈSE

pour obtenir le grade de

Docteur de l'École Normale Supérieure de Lyon
spécialité : Informatique

au titre de l'école doctorale de MathIF

présentée et soutenue publiquement le 13 décembre 2005

par Mademoiselle Hélène RENARD

Équilibrage de charge et redistribution de données sur plates-formes hétérogènes

Directeurs de thèse : Monsieur Yves ROBERT
Monsieur Frédéric VIVIEN

Après avis de : Monsieur Jacques BAHI, Rapporteur
Monsieur Michel DAYDÉ, Rapporteur

Devant la commission d'examen formée de :

Monsieur Jacques BAHI, Rapporteur
Monsieur Michel DAYDÉ, Rapporteur
Monsieur Serge MIGUET, Extérieur
Monsieur Yves ROBERT, Directeur de Thèse
Monsieur Denis TRYSTRAM, Extérieur
Monsieur Frédéric VIVIEN, Directeur de Thèse

Les mots manquent aux émotions,
Victor Hugo, *Le Dernier jour d'un condamné* (1802-1885).

Certaines personnes aussi ;
Pour toi, qui ne sera pas là ...

Soyons reconnaissants aux personnes qui nous donnent du bonheur ; elles sont les charmants jardiniers par qui nos âmes sont fleuries.
Marcel Proust (1871-1922).

Merci !

Le seul moyen de se délivrer d'une tentation, c'est d'y céder paraît-il ! Alors j'y cède en disant en grand Merci aux personnes qui ont cru en moi et qui m'ont permis d'arriver au bout de cette thèse.

Le premier grand Merci ira à mes deux chefs, Yves Robert (le gentil chef) et Frédéric Vivien (le méchant chef) sans qui tout ce travail n'aurait pas été possible ! Les débuts ont certes été difficiles : premier échange un peu dur entre une amiènoise toute timide qui ne connaissait pas grand chose à l'informatique parallèle et un chef qui lui dit « gentiment » de venir jusqu'à son bureau plutôt que d'envoyer un mail ! Mise en application directe ... et mise à contribution aussi ! Surtout lorsqu'il s'agissait de corriger mes erreurs de blonde, comme par exemple, des tabulations à la place d'espaces dans un makefile ou bien encore la compilation du mauvais programme !
Merci à toi Yves pour avoir eu confiance en moi et pour avoir si bien joué ton rôle de « Papa Yves ». Merci à toi Frédo pour ta rigueur, même si j'ai détesté bon nombre de fois ton stylo rouge ! Merci à vous deux pour votre patience pendant ces trois années de thèse et surtout Merci pour votre compréhension et votre présence pendant les moments difficiles.

Le second grand Merci ira à Jacques Bahi, Michel Daydé, Serge Miguet et Denis Trystram qui ont accepté d'être membres de mon jury de thèse. Et plus particulièrement, Merci à Jacques et Michel pour leurs remarques constructives sur la première version de cette thèse.

Un énorme Merci à tous mes co-bureaux qui m'ont supportée pendant ces trois ans et qui m'ont convaincue que Linux était mieux que Windows ! Merci à Antoine et Loris d'avoir écouté patiemment toutes mes p'tites histoires et surtout Merci à toi Antoine pour ta présence dans les bons comme les mauvais moments. Merci à Emmanuel de m'avoir « débauchée » en m'emmenant dans les concerts de ska et en me faisant découvrir le Barberousse !

Un grand Merci à mes voisins de bureau : Merci à toi Arnaud pour ta disponibilité et ta patience ; Merci à toi Abdou pour m'avoir fait découvrir Dark Tranquillity et pour m'avoir suivie jusqu'à Montpellier pour voir Mickaël Stanne ! Mais surtout Merci pour tout ce que tu as fait pendant la dernière ligne droite : tu n'as jamais baissé les bras même lorsque je faisais mon mauvais caractère !

Un Merci particulier pour toi Fred, pour ton écoute et pour l'entrée en trombe au Parc des Princes avec les premières notes de Slipknot !

Un autre Merci particulier à Anne-Pascale, Corinne, Isa et Sylvie pour leur bonne humeur et le sourire qu'elles n'ont jamais perdu même après deux changements de billets d'avion !

Le Merci suivant s'adresse aux personnes qui ont compté pour moi à leur manière, en dehors du cadre du travail et qui ont fait que j'en suis là aujourd'hui : Merci à toi Guillaume pour le petit bout de route que tu as fait avec moi d'Amiens à Lyon ; Merci à toi Romain pour ta force de caractère et tes bisous sur le nez ! Merci à vous, Amandine et Pascal pour nos folles soirées jeux ! Merci à toi Laurent pour m'avoir fait découvrir la ville qui ne dort jamais et les *taxis dog* ! Et enfin Merci à toi copain et à ton p'tit bonhomme pour avoir fait briller le soleil en hiver !

Le dernier Merci sera pour ma famille et surtout pour mes parents qui ont laissé partir leur fille ; Merci pour avoir fait de moi ce que je suis aujourd'hui.

Une pensée pour terminer ces remerciements pour toi qui n'a pas vu l'aboutissement de mon travail mais je sais que tu en aurais été très fier de ta petite-fille !

Table des matières

Chapitre 1

Introduction

La victoire appartient à celui qui y croît le plus, et moi j'y crois !
Citation amiénoise

Les besoins en puissance de calcul vont en s'accroissant dans une multitude de domaines (simulation et modélisation, traitement du signal, d'images, fouille de données, télé-immersion, etc.) et le parallélisme est une tentative de réponse toujours d'actualité. Partant du vieil adage selon lequel « l'union fait la force », dès les années 60, les premiers super-calculateurs ont fait leur apparition et ont connu leur heure de gloire jusque dans les années 90. L'effondrement des sociétés commercialisant les super-calculateurs s'est réalisé du fait de l'avènement d'architectures de type grappes de stations, bien moins coûteuses. Cette évolution n'a été possible que grâce à des efforts constants en terme de conception et de développement logiciel.

Les plates-formes de calcul parallèle ne se résument donc plus aux super-calculateurs monolithiques de Cray, IBM ou SGI. Les réseaux hétérogènes de stations (les machines parallèles du « pauvre » par excellence) sont monnaie courante dans les universités ou les entreprises. Faire coopérer un grand nombre de processeurs du commerce, agencés en grappes et reliés par un réseau plus ou moins rapide, s'avère nettement moins onéreux mais aussi nettement plus délicat. Ajoutons à cela que la tendance actuelle en matière de calcul distribué est à l'interconnexion de machines parallèles classiques et de grappes, réparties à l'échelle d'un pays ou d'un continent, par des liens rapides afin d'agréger leur puissance de calcul. Il apparaît donc clairement que l'hétérogénéité est une caractéristique essentielle des plates-formes de calcul d'aujourd'hui et de demain. La conception d'algorithmes adaptés – ou l'adaptation d'algorithmes existants – à ces nouveaux environnements parallèles est donc nécessaire et a motivé les travaux présentés dans cette thèse.

Algorithmes itératifs

« Un algorithme itératif a pour but de construire, à partir d'un vecteur initial, une suite de vecteurs, dont la limite est la solution du problème considéré. L'avènement des calculateurs parallèles a donné à ce domaine de recherche une nouvelle vitalité. Il ne s'agit pas d'implanter naïvement, sur calculateurs parallèles, un algorithme itératif séquentiellement convergent, mais d'étudier les problèmes de convergence de l'algorithme parallèle, sa meilleure parallélisation et sa réelle efficacité » [18].

Les modèles itératifs

Les méthodes itératives résolvant les systèmes linéaires creux et de grande taille gagnent du terrain dans de nombreux domaines de calcul scientifique. Jusqu'à présent, les méthodes directes étaient souvent préférées aux méthodes itératives pour les applications réelles du fait de leur robustesse et de leur comportement prévisible. Ces méthodes, bien que très performantes, utilisent une quantité de mémoire significative, qui peut souvent dépasser la mémoire disponible sur la plate-forme considérée. De plus, un nombre de solveurs efficaces itératifs ont été découverts et le besoin croissant de résoudre des systèmes linéaires très grands a déclenché un décalage apparent et rapide vers des techniques itératives dans de nombreuses applications. Elles constituent donc une très bonne alternative lorsque les méthodes directes ne sont pas envisageables compte tenu des limitations matérielles de la machine cible (mémoire, puissance de calcul), et lorsque l'application visée ne nécessite pas une solution « très » précise (itérations non linéaires, incertitude sur les données, etc.). Dans ce dernier cas, le critère d'arrêt des méthodes itératives permet de contrôler le niveau de précision souhaité. De plus, ces méthodes itératives gagnent du terrain parce qu'il est plus facile de les implémenter efficacement sur des ordinateurs haute performance que les méthodes directes. Parmi les méthodes itératives connues et très utilisées, il y a celle du gradient conjugué. Elle présente l'avantage d'être facilement programmable (toutefois, il existe d'autres méthodes plus efficaces) et d'être susceptible d'une extension au cas où l'équation de mesure n'est pas parfaitement linéaire. Un exemple d'utilisation de la méthode du gradient conjugué peut être le couplage d'un algorithme de type gradient conjugué préconditionné appliqué au champ de déplacement mécanique [23]. Dans cet article, les auteurs présentent un nouvel algorithme de couplage entre l'écoulement des fluides en milieux poreux du simulateur de réservoir et le code de géomécanique modélisant la compaction du milieu poreux. Ils concluent que l'algorithme de gradient conjugué est bien plus robuste et converge plus rapidement que l'algorithme décalé, avec un coût supplémentaire par itération qui reste négligeable en pratique. Ce simple exemple, et il en existe de nombreux dans la littérature, nous prouve l'efficacité des méthodes itératives.

Les algorithmes itératifs et leurs applications

Intéressons nous de plus près aux méthodes itératives et plus particulièrement aux algorithmes itératifs et à leurs applications.

Il existe deux types d'algorithmes itératifs : synchrone et asynchrone. Examinons les différences entre ces deux catégories : pour les algorithmes itératifs synchrones, les processeurs commencent la même itération au même moment alors que pour les algorithmes itératifs asynchrones, les processeurs ne calculent pas forcément la même itération à un instant t, ce qui permet de ne plus avoir de période d'inactivité. Considérons un exemple : le problème d'Akzo-Nobel médical. Il s'agit de la modélisation de la pénétration d'anticorps dans un système cancéreux sur une dimension, décrit par une équation aux dérivées partielles, où nous cherchons la concentration d'anticorps dans le tissu (notée y). La modélisation de ce système se fait par une onde y, dépendant au maximum des deux précédentes et des deux suivantes. À chaque itération de l'algorithme (décrit dans [1]), un schéma « Euler implicite » est utilisé pour approcher la dérivée partielle en temps, puis l'algorithme de Newton afin de résoudre le système non-linéaire résultant ainsi que le calcul de chaque onde sur tout l'intervalle de temps. Cette étape est appelé « solve » dans l'algorithme itératif global. Cet algorithme itératif est donc décrit de la manière suivante : pour chaque composante y indicée par j, nous avons $y_{new}[j] = solve(y_{old}[j])$. De plus, si le voisin de gauche existe, un envoi asynchrone de deux premières ondes est effectué. Il en est de même pour le voisin de droite avec les deux dernières ondes. Puis y_{new} devient y_{old}. Ces étapes sont répétées tant que la convergence globale n'est pas atteinte.

Une autre façon d'utiliser les algorithmes itératifs peut se faire sur des séquences vidéo [3]. Prenons l'exemple de la construction d'un panorama à partir d'une séquence vidéo provenant d'une caméra subissant un mouvement de rotation pure. La construction de panorama passe par le calcul du mouvement de la caméra entre images consécutives de la séquence. Ce calcul est basé sur la minimisation d'une fonction de coût pouvant être de deux types différents. Les méthodes denses sont basées sur une erreur exprimée en fonction de la différence d'intensité entre tous les pixels correspondants des deux images. Les méthodes creuses sont basées sur une erreur exprimée en fonction de la distance entre certains pixels correspondants. En ce qui concerne les méthodes denses, des algorithmes sont proposés dans [42, 64].

Nous venons de voir sur un exemple l'utilité des algorithmes itératifs dans le domaine de la vidéo. Un autre secteur lui aussi assez parlant, où sont utilisés ces algorithmes itératifs, est le domaine des communications numériques. Afin d'améliorer les performances des récepteurs de télécommunications numériques, les systèmes de transmission peuvent être développés à l'aide d'une optimisation conjointe [60] (par exemple égalisation et décodage conjoints, modulation et décodage...) pour ne plus optimiser séparément et de manière antagoniste les différents éléments de la chaîne de transmission. Si le récepteur optimal associé au système conjoint est de complexité trop importante, il existe une solution sous-optimale de complexité réduite maintenant de bonnes performances. Les algorithmes itératifs ont été un des progrès majeurs en

communications numériques ces dernières années. Leur principe consiste à itérer des « estimations souples » des quantités à transmettre entre deux étapes de l'algorithme de réception. Ceci est rendu nécessaire par l'impossibilité d'effectuer le traitement en une seule étape (trop de complexité), ainsi que le besoin que chacun des blocs influence le fonctionnement de l'autre (pour améliorer les performances). Cette procédure a été étudiée tout d'abord dans le contexte du codage canal (on itère le décodage de deux codes simples, combinés en un code produit), de la turbo-égalisation (on itère le décodage canal et l'égalisation de type MAP), et a récemment été appliqué à une situation de type turbo-détection des modulations à bits entrelacés (BICM) [28].

Modélisation de plates-formes hétérogènes

La plupart des résultats d'ordonnancement que l'on peut trouver dans la littérature sont obtenus grâce à des hypothèses assez restrictives et à des modèles assez simples : les réseaux d'interconnexion sont souvent très simplistes en regard de la réalité ; les ressources de calcul et de communication sont souvent supposées ne pas avoir de variation de performances ; il est toujours possible d'obtenir des prédictions parfaites des vitesses des ressources. Ces hypothèses simplificatrices sont nécessaires à la compréhension de certains phénomènes mais ne sont jamais vérifiées en pratique et les heuristiques qui en découlent sont rarement confrontées à la réalité.

S'il est difficile de garantir les performances d'une heuristique, il est également difficile de valider expérimentalement son efficacité. Cette étape est pourtant souvent nécessaire pour pouvoir comparer objectivement ses algorithmes à ceux proposés dans la littérature. S'il était possible de recourir à des expériences grandeur nature quand on se plaçait dans un cadre homogène, ce n'est plus le cas dans un cadre hétérogène. En effet, de telles expériences sont très délicates à mener en raison de l'instabilité latente des plates-formes de calcul hétérogène et distribuée. Il est impossible de garantir que l'état d'une plate-forme de calcul qui n'est pas entièrement dédiée à l'expérimentation va rester le même entre deux expériences, ce qui empêche donc toute comparaison rigoureuse. On utilise donc des simulations afin d'assurer la reproductibilité des expériences, toute la difficulté étant alors d'arriver à simuler un tel environnement de façon réaliste.

Pour parvenir à une comparaison honnête des algorithmes, une approche efficace consiste à effectuer des simulations utilisant des traces, c'est-à-dire utilisant des enregistrements de différents paramètres d'une plate-forme réelle pour obtenir un comportement réaliste. SIMGRID [54], le logiciel que nous avons utilisé, a été développé par Arnaud Legrand, lorsqu'il était à l'École normale supérieure de Lyon, en collaboration avec Henri Casanova de l'Université de Californie, San Diego. C'est un simulateur modulaire permettant de simuler une application distribuée où les décisions d'ordonnancement peuvent être prises par différentes entités. La force de ce simulateur réside dans sa capacité à importer et simuler aisément des plates-formes réalistes (du réseau de stations de travail à la grille de métacomputing).

Contexte de la thèse

La distribution des calculs (ainsi que des données associées) sur plate-forme parallèle hétérogène peut être effectuée soit dynamiquement, soit statiquement. Une grande partie de la littérature traite de stratégies dynamiques, qui nécessitent des phases périodiques de ré-allocation pour remédier au déséquilibre observé. De manière générale, il y a une difficulté certaine à déterminer un compromis entre les paramètres de distribution des données et les politiques de génération de processus et de migration. Des calculs redondants peuvent être nécessaires pour utiliser une plate-forme au meilleur de ses capacités. Si les caractéristiques de la plate-forme cible (vitesses des processeurs et capacités des liens) et de l'application cible (coût des calculs et des communications associés à chaque partition de données) sont connues avec suffisamment de précision, alors un excellent niveau de performance peut être atteint par le biais de stratégies statiques. Toutefois, des schémas sophistiqués de distribution de données sont indispensables pour atteindre ce niveau de performance. C'est suite à ces observations qu'il nous est apparu nécessaire de revisiter les problèmes de placement, d'allocation et de ré-allocation, entre autres.

Nous nous sommes donc intéressée à la mise en œuvre d'algorithmes itératifs sur des grappes hétérogènes. Ces algorithmes fonctionnent avec un volume important de données (calcul de matrices, traitement d'images, etc.), qui sera réparti sur l'ensemble des processeurs. À chaque itération, des calculs indépendants sont effectués en parallèle et certaines communications ont lieu. Prenons l'exemple d'une matrice rectangulaire de données : l'algorithme itératif fonctionne répétitivement sur cette matrice, divisée en tranches verticales (ou horizontales) allouées aux processeurs. À chaque étape de l'algorithme, les tranches sont mises à jour localement et les informations frontières sont échangées entre tranches consécutives. Cette contrainte géométrique implique que les processeurs soient organisés en anneau virtuel. Chaque processeur communique seulement deux fois, une fois avec son prédécesseur (virtuel) dans l'anneau et une fois avec son successeur. Il n'existe pas de raison *a priori* de réduire le partitionnement des données à une unique dimension et de ne l'appliquer que sur un anneau de processeurs unidimensionnel. Cependant, un tel partitionnement est très naturel et nous montrerons que trouver l'optimal est déjà très difficile.

Contribution et plan de la thèse

Placement et équilibrage de charge

Dans un premier temps (chapitre 3), nous nous sommes intéressée à l'équilibrage de charge sur plate-forme hétérogène. Nous nous sommes tout d'abord placée sur un réseau hétérogène complet, composé de processeurs ayant des vitesses de calcul différentes, communicant par des liens de bandes passantes différentes. Nous pouvons alors nous placer dans le cas de figure « simple » où il n'y aura pas de partage de liens.

Sans partage de liens

D'un point de vue architectural, le problème se décompose en deux parties : (i) sélectionner les processeurs participant à la solution ; (ii) décider de leur position dans l'anneau. Tout cela, de manière à ce que le temps total d'exécution soit minimal. Nous avons ensuite montré que ce problème est un problème NP-complet (section 3.3.4). Nous avons mis en œuvre une heuristique (l'heuristique gloutonne) qui commençe par sélectionner le processeur le plus rapide, puis, qui itérativement ajoute un nouveau processeur dans la solution de manière à minimiser notre fonction objectif. Une autre méthode est la résolution exacte par programmation linéaire en entiers (ILP) à l'aide de pipMP [31]. Nous comparons alors nos deux méthodes, à savoir la résolution linéaire en entiers et l'heuristique gloutonne.

Ces travaux ont fait l'objet de publications dans [A, C].

Avec partage de liens

La seconde partie de ce travail sur l'équilibrage de charge a consisté à considérer un réseau totalement hétérogène, mais cette fois non complet. Nous sommes donc dans le même cas de figure que précédemment mais sans l'hypothèse de complétude. Une difficulté majeure est alors que plusieurs communications peuvent utiliser le même lien physique, les réseaux de communication de grappes hétérogènes n'étant pas totalement connectés. Si plusieurs communications partagent le même lien physique, nous décidons, dans notre heuristique, quelle sera la fraction de bande passante attribuée à chaque communication.

Une fois l'anneau et le routage décidés, il reste à déterminer le meilleur partitionnement des données. La qualité de la solution finale dépend alors d'un grand nombre de paramètres de l'application ainsi que de l'architecture, et le problème d'optimisation est naturellement difficile à résoudre. Nous avons montré que ce problème est un problème NP-complet. Nous avons alors développé et implémenté une heuristique gloutonne qui prend en compte le partage des liens. Cette heuristique étant simpliste, nous lui avons apporté deux améliorations : le *Max-Min fairness* [7] et la résolution quadratique en utilisant le logiciel KINSOL [65].

L'impact du partage de liens

Pour évaluer l'impact du partage de la bande passante des liens, nous avons retravaillé avec la version simple du problème où nous voyions le réseau comme un graphe complet : entre chaque paire de nœuds, le routage est fixé (plus court chemin en terme de bande passante), et la bande passante est définie par le lien le plus lent dans le chemin routé. Ce modèle simplifié nous permet d'obtenir un anneau de processeurs en ne tenant pas compte du partage de liens (cet anneau étant trop optimiste) ainsi qu'un temps d'exécution. Ce même anneau est alors donné à la seconde partie de

notre heuristique gloutonne, qui va calculer l'allocation de bandes passantes en prenant en compte le partage de liens. Au final, nous obtenons un anneau dont les liens sont partagés (anneau réaliste) et un nouveau temps d'exécution que nous comparons au premier temps d'éxécution obtenu. Par ce biais, nous obtenons une manière commode d'évaluer l'impact des différentes hypothèses faites sur les communications.

Les conclusions pouvant être tirées de ces expérimentations sont les suivantes :

- Lorsque l'impact du coût de communication est faible, le but principal est d'équilibrer les calculs et les deux heuristiques (avec et sans partage de liens) sont équivalentes.
- Lorsque le rapport communication/travail devient plus important, l'effet de la contention des liens devient évident et la solution retournée par l'heuristique avec partage de liens est bien meilleure.

Ces travaux ont fait l'objet de publications dans [A, D, E, G].

Redistribution de données

Dans un deuxième temps, à cause des variations des ressources ou des besoins de l'application, les données ont besoin d'être redistribuées sur l'ensemble des processeurs participants afin que la charge de travail reste équilibrée (chapitre 4). C'est pourquoi, nous nous sommes intéressée au problème de redistribution de données sur des anneaux de processeurs homogènes et hétérogènes. Ce problème surgit dans plusieurs applications, après chaque phase d'équilibrage de charge.

Nous nous sommes placée dans le cadre de travail suivant : nous supposons que les processeurs participant à la solution sont organisés en anneau, unidirectionnel ou bidirectionnel, et ayant des liens de communication homogènes ou hétérogènes (nous avons donc traité quatre cadres de travail). Chaque processeur possède dès le départ un certain nombre de données, puis le système (l'oracle) décide que chaque processeur est surchargé ou sous-chargé. Le but est alors de déterminer les communications nécessaires afin de rétablir l'équilibre (c'est ce que nous avons appelé la redistribution de données) et d'effectuer ces communications en un temps minimal.

Anneau unidirectionnel

Dans un premier temps, nous nous sommes placée dans le cadre d'un anneau unidirectionnel homogène ; c'est-à-dire qu'un processeur P_i ne peut envoyer des données qu'à son successeur P_{i+1} et les liens de communication ont une capacité constante. Nous avons obtenu une borne inférieure du temps d'exécution de la redistribution de données ainsi qu'un algorithme optimal. Nous avons mis en œuvre cet algorithme en C afin de vérifier nos hypothèses puis nous avons prouvé son exactitude ainsi que son optimalité.

Dans un second temps, nous nous sommes placée dans le cadre d'un anneau unidirectionnel hétérogène ; c'est-à-dire dont les liens de communication n'ont plus même capacité. Nous avons, de la même manière que précédemment, obtenu un algorithme optimal. Cependant, du fait de l'hétérogénéité des liens de communication, nous obtenons un algorithme asynchrone. Cette dernière caractéristique implique l'exactitude de notre algorithme par construction ; nous n'avons eu qu'à prouver son optimalité.

Le cas unidirectionnel a donc été complètement résolu grâce à nos algorithmes optimaux et aux preuves que nous avons fournies.

Anneau bidirectionnel

La seconde étape a été de considérer les anneaux bidirectionnels. Nous avons tout d'abord considéré un anneau bidirectionnel homogène, c'est-à-dire dont les liens ont les mêmes capacités mais où un processeur peut envoyer des données à ses deux voisins dans l'anneau. Nous avons procédé de la même manière que dans le cas de l'anneau unidirectionnel homogène : nous avons établi une borne inférieure du temps d'exécution de la redistribution de données ainsi qu'un algorithme optimal qui atteint cette borne. Nous avons mis en œuvre cet algorithme en C et nous l'avons testé intensivement afin de vérifier nos hypothèses puis nous avons prouvé son exactitude ainsi que son optimalité.

Nous nous sommes intéressée ensuite à un anneau bidirectionnel hétérogène, c'est-à-dire, le cas général. Nous n'avons par contre pas obtenu d'algorithme optimal dans ce cas de figure. Cependant, en supposant que chaque processeur possède initialement les données nécessaires à envoyer pendant l'exécution de l'algorithme (principe de la redistribution légère), nous sommes capable d'obtenir une solution optimale. Si par contre, l'hypothèse de redistribution légère n'est pas réalisable, nous avons une borne inférieure du temps d'exécution de la redistribution de données mais nous ne savons pas si cette borne est toujours atteignable. Cependant, nous n'avons pas de contre-exemple comme quoi cette borne n'est pas toujours atteignable. Le problème reste donc ouvert dans le cas général et la complexité de la borne inférieure montre que ce problème est ardu.

L'impact de la redistribution de données

Pour évaluer l'impact de la redistribution de données, nous avons utilisé le simulateur SIMGRID [54] afin de modéliser une application itérative sur une plate-forme générée par Tiers [13, 26]. Nous avons alors pu remarquer que plus la puissance de calcul était grande, moins les redistributions étaient nécessaires. Et inversement, moins la puissance de calcul est importante, plus les redistributions de données sont nécessaires ; cependant, il ne faut pas non plus faire trop de redistributions.

Nous proposons donc des algorithmes qui visent à optimiser la redistribution de données pour des anneaux unidirectionnels et bidirectionnels, et nous donnons toutes

les preuves de correction de ces algorithmes. Une des contributions principales de cette étude sur la redistribution de données est que nous pouvons prouver l'optimalité des algorithmes proposés dans tous les cas, sauf dans le cas d'un anneau hétérogène bidirectionnel, pour lequel le problème reste ouvert.

Ces travaux ont fait l'objet de publications dans [B, F, H].

Perspectives

Prise en compte des différentes variations : performances de la plate-forme et nombre de nœuds de l'anneau solution

En dernier lieu (chapitre 5), dans cette thèse, nous nous sommes intéressée aux différentes variations qui peuvent avoir lieu au cours du temps : variation de performances des plates-formes hétérogènes et variation du nombre de nœuds de l'anneau solution. Nous entendons par variations de performances de la plate-forme, le changement des caractéristiques de cette plate-forme (vitesses de calcul des processeurs et capacités des bandes passantes). Cela va donc influer sur le nombre de nœuds de l'anneau solution. Comme dans les chapitres précédents, à chaque itération, des calculs indépendants sont effectués en parallèle, et des communications ont lieu entre les processeurs consécutifs dans l'anneau. Lorsque les caractéristiques de plate-forme changent, il peut être intéressant de mettre à jour la solution courante (attribution d'anneau et de données). Nous avons commencé à étudier plusieurs stratégies pour résoudre ce problème : en particulier nous avons discuté les mises à jour locales avec ou sans la redistribution intermédiaire de données, et les changements globaux de la solution.

L'objectif principal de ce travail est d'étudier un problème d'optimisation bien plus complexe que précédemment puisque les paramètres de la plate-forme changent de manière significative pendant l'exécution de l'algorithme itératif. La solution courante (anneau virtuel, attribution de bande passante, distribution de données) peut ne plus convenir aux nouveaux paramètres de la plate-forme. Le problème est alors de décider *quand* et *comment* changer la solution courante.

Il existe plusieurs *stratégies de transition* possibles pour mettre à jour la solution courante. Nous pouvons simplement garder la solution courante et chercher une meilleure redistribution de données dans l'anneau. Si nous en trouvons une, nous redistribuons les données. Nous pouvons aussi construire un nouvel anneau. Par exemple il peut être nécessaire de remplacer un ou plusieurs processeurs qui sont devenus trop lents.

Ce travail est pour l'instant théorique, il est en cours de validation expérimentale. Mais les résultats obtenus dans les chapitres 3 et 4, nous conduisent à penser que notre approche s'avèrera pertinente.

Chapitre 2

Position du problème

L'ordinateur parfait a été inventé : on entre un problème et il n'en ressort jamais !
Al Goodman

La littérature étant très riche au niveau de l'équilibrage de charge (dynamique ou bien statique) ainsi qu'au niveau redistribution de données et variation de performances, nous allons nous contenter dans ce chapitre de citer quelques articles (liste non exhaustive) afin de donner au lecteur une vue d'ensemble du problème.

2.1 Équilibrage de charge

Le problème de l'équilibrage de charge consiste à allouer des données au départ (fractions d'une application donnée), puis éventuellement à les redistribuer sur un ensemble de processeurs afin de minimiser leur temps de traitement. Ce problème peut être classifié en deux sous-catégories : l'équilibrage de charge dynamique et l'équilibrage de charge statique. Bien que nous étudierons principalement l'équilibrage de charge statique dans cette thèse, nous commençons par résumer quelques travaux sur l'équilibrage de charge dynamique.

2.1.1 Approche dynamique

La littérature étant vaste en ce qui concerne l'équilibrage de charge dynamique, nous allons voir des approches différentes pour traiter ce problème.

Il faut, toutefois, commencer par préciser le sens de l'équilibrage de charge dynamique dans la littérature, puisque de nombreuses interprétations sont possibles. Par exemple, nous pouvons parler d'équilibrage de charge dynamique lorsque la charge varie pendant le processus d'équilibrage de charge [22]. Nous parlerons aussi d'équilibrage de charge dynamique lorsque la topologie de la plate-forme varie [2]. Dans ce

dernier exemple, J. Bahi, R. Couturier et F. Vernier voient le réseau dynamique comme un réseau ayant des liens dynamiques. Ils supposent qu'aucun ordinateur ne peut être ajouté ou définitivement retiré de ce réseau et que chaque nœud connaît l'ensemble des arêtes viables (des arêtes peuvent être perdues lors de l'exécution de l'algorithme suite à des communications fautives ou bien encore des *timeout*). Avec ces considérations, les trois algorithmes qu'ils proposent sont synchrones et distribués. Leur premier algorithme est un algorithme d'équilibrage de charge par diffusion classique, nécessitant quelques adaptations du fait de la dynamicité du réseau (la matrice de diffusion intègre l'information lorsqu'un lien est manquant). Les deux autres (GAE et l'algorithme de diffusion relaché accélérant la convergence de la diffusion classique) sont des adaptations de cet algorithme. Le modèle de GAE peut être vu comme un modèle de diffusion dans lequel, au temps t et pour chaque processeur, toutes les arêtes exceptée une sont inutilisables (le choix de l'unique voisin de chaque processeur s'effectue à l'aide de stratégies arbitraires, aléatoires ou plus sophistiquées). Quant au dernier algorithme, il est basé sur le schéma par diffusion, en introduisant un paramètre de relaxation dans le but d'accélérer la convergence. Les auteurs simulent ensuite différents réseaux avec un certain pourcentage d'arêtes inutilisables (de 0% à 50%) pour illustrer le comportement de ces algorithmes et pour accentuer leur convergence vers la distribution uniforme de la charge de travail. Les algorithmes de J. Bahi, R. Couturier et F. Vernier permettent donc d'équilibrer la charge sur un réseau dynamique dans lequel les liens de communication ne sont pas fiables à 100%.

Une autre approche pour l'équilibrage de charge dynamique est d' « utiliser le passé pour prédire l'avenir », c'est-à-dire d'utiliser la vitesse de calcul observée pour chaque processeur ou ressource afin de décider de la redistribution du travail. De nombreux auteurs [16, 17, 6] se sont intéressés à cette approche, et notamment dans [17] où l'équilibrage de charge implique d'assigner à chaque processeur un travail proportionnel à ses capacités, en minimisant le temps d'exécution du programme. Bien que l'équilibrage de charge statique puisse résoudre beaucoup de problèmes (par exemple ceux provoqués par l'hétérogénéité des processeurs) pour la plupart des applications régulières, la charge passagère due aux utilisateurs multiples sur un réseau de stations de travail nécessite une approche dynamique afin d'équilibrer la charge. Les auteurs examinent les différents comportements des stratégies d'équilibrage de charge : global contre local et centralisé contre distribué. Ils montrent ainsi que chaque stratégie est la meilleure pour différentes applications données en fonction des variations des paramètres du programme et du système. Par conséquent, un équilibrage de charge personnalisé devient essentiel pour obtenir de bonnes performances. Un processus hybride de modélisation et de décision, choisissant la meilleure méthode d'équilibrage de charge est présenté. Ce processus fait appel à une génération automatique de code parallèle et à une librairie d'exécution pour l'équilibrage de charge. Ainsi, les auteurs montrent qu'il est possible de personnaliser un modèle d'équilibrage de charge pour un programme avec différents paramètres.

Dans cette approche, tout est basé sur le « passé ». Il existe bien sûr d'autres moyens de faire de l'équilibrage de charge dynamique et notamment en faisant d'autres sup-

positions.

Une vision très particulière de l'équilibrage de charge dynamique est basée sur la redistribution de données parmi les processeurs participants pendant l'exécution de l'algorithme. Cette redistribution est faite en transférant des données des processeurs les plus chargés vers les processeurs les moins chargés. Cette phase d'équilibrage de charge peut être centralisée par un seul processeur ou être distribuée sur l'ensemble des processeurs. Par exemple, dans [37], le problème principal est le changement permanent de la charge de chaque station de travail (les auteurs ont travaillé sur une architecture organisée en clusters), ce qui rend imprévisible le temps d'exécution des applications. En ce qui concerne le modèle de simulation, les auteurs ont utilisé le schéma suivant : une seule file d'attente et un seul serveur.

Un autre exemple des nombreux modèles existants pour les stratégies d'équilibrage de charge dynamiques dans [67] où H. Willebeeck-Lemair et P. Reeves présentent cinq approches de manière à illustrer la différence entre « connaissance » (exactitude de chaque décision d'équilibrage) et « surcoût » (quantité de processus et communication supplémentaires induite par la phase d'équilibrage). Une des cinq approches se résume de la manière suivante : tous les processeurs informent leurs voisins proches de leur niveau de charge et mettent à jour ces informations pendant l'exécution de l'application. Ainsi, en définissant un certain seuil (seuil qui permet de définir la surcharge ou sous-charge d'un processeur), les processeurs surchargés vont pouvoir envoyer des données aux processeurs sous-chargés. Cette stratégie s'appelle « Sender Initiated Diffusion (SID) ». Nous ne décrirons pas ici la RID « Receiver Initiated Diffusion » ; nous allons simplement énoncer les différences. La stratégie RID diffère de la SID au niveau de la phase de migration de données : supposons qu'un processeur sous-chargé envoie en premier des requêtes de demande de charge. Il reçoit alors les réponses à ses requêtes. Par conséquent, des messages supplémentaires sont envoyés pour chaque transfert de donnée. La RID va alors nécessiter un nombre de messages plus grand que la SID. Dans ces deux stratégies, le nombre total d'itérations nécessaires pour atteindre l'équilibrage global dépend de l'application ainsi que de la topologie. Les trois autres stratégies utilisées par les auteurs sont le modèle gradient, qui utilise une carte de routage afin de faire migrer les données des processeurs les plus chargés vers les processeurs les moins chargés et les plus proches, la méthode d'équilibrage hiérarchique, qui organise le système en une hiérarchie de sous-systèmes, et enfin, la méthode d'échange dans les différentes dimensions d'un hypercube, qui nécessite une phase de synchronisation pour équilibrer la charge puis équilibrer itérativement ensuite.

Revenons à une vision un peu plus généraliste de l'équilibrage de charge dynamique, c'est-à-dire avec des suppositions moins restrictives. Plusieurs auteurs ([16], [21]) s'intéressent à l'équilibrage de charge dynamique pour des applications parallèles. Par exemple, J. Weissman examine dans [21] le problème de l'adaptation d'applications parallèles de données dans un environnement dynamique de stations de travail. Il développe un cadre de travail analytique pour comparer un large éventail de stratégies : équilibrage de charge dynamique, déplacement de données, ajout

ou suppression de processeurs. Ces stratégies ont été évaluées afin d'en déterminer leur coût et bénéfice pour trois applications parallèles représentatives : un solveur jacobien pour l'équation de Laplace, l'élimination gaussienne avec pivot partiel et une application de comparaison de séquence de gènes. Il a trouvé que le coût et le bénéfice de chaque méthode peuvent être prédit avec une grande précision (10%) pour toutes ces applications. Cela montre que le cadre de travail est applicable à une large variété d'applications parallèles. Il montre ensuite que cette prédiction permet la sélection dynamique de la méthode la plus appropriée. Grâce à la librairie développée, le gain de performance pour les trois applications va de 25% à 45%. De plus, il va à l'encontre de l'idée préconçue que le déplacement de données est coûteux et montre que celui-ci peut être bénéfique même dans le cas d'applications parallèles impliquant des coûts de communications non négligeables.

De même dans [16], où M. Cierniak, M. Zaki et W. Li s'attaquent au problème d'ordonnancer des boucles parallèles pour des réseaux hétérogènes de stations de travail. Différents aspects de l'hétérogénéité dans la programmation parallèle, tels que le programme, les processeurs, la mémoire ainsi que le réseau sont pris en compte. Un programme hétérogène contient des boucles parallèles avec une quantité de travail différente à chaque itération ; des processeurs hétérogènes ont des vitesses de calcul différentes ; une mémoire hétérogène se rapporte à la quantité de mémoire utilisateur disponible sur les machines ; et un réseau hétérogène a des coûts de communication différents entre les processeurs. Les auteurs proposent un modèle simple mais pourtant complet pour l'utilisation en compilation dans un réseau de processeurs. De plus, ils développent des algorithmes afin de produire des ordonnancements de boucles optimaux pour l'équilibrage de charge, les optimisations de communication, la contention des liens dans le réseau ainsi que l'hétérogénéité de la mémoire. Afin de modéliser les processeurs hétérogènes, ils ont introduit un paramètre : la vitesse normalisée du processeur, qui est étroitement liée aux paramètres de l'application. Leurs expériences montrent que ces nouvelles techniques améliorent significativement l'exécution des boucles parallèles par rapport aux méthodes existantes.

D'autres auteurs, comme Z. Lan et V. Taylor s'intéressent eux aussi à l'équilibrage de charge dynamique mais cette fois sans l'hypothèse d'applications parallèles comme précédemment. Ils proposent un modèle d'équilibrage de charge dynamique pour systèmes distribués dans [52]. Leur modèle prend en compte l'hétérogénéité des processeurs ainsi que l'hétérogénéité et la charge dynamique du réseau. L'hétérogénéité des processeurs est traitée en leur attribuant un poids relatif à leur vitesse de calcul. La charge de travail est donc distribuée en fonction de ces poids. Afin de traiter l'hétérogénéité du réseau, le processus d'équilibrage de charge est divisé en deux phases : une globale et une locale. Le premier objectif est de minimiser les communications distantes ainsi que d'équilibrer efficacement la charge sur les processeurs. Une des issues clé de la redistribution globale est de décider quand une action doit être effectuée et si c'est avantageux. La décision doit être rapide et basée sur des modèles simples. Une heuristique est donc proposée par les auteurs pour évaluer le gain et le coût de la redistribution lors de la phase globale. Leurs expériences illustrent les

avantages de leur équilibrage de charge dynamique à manipuler l'hétérogénéité et la charge dynamique du réseau. Elles montrent qu'en utilisant le modèle d'équilibrage de charge dynamique distribué, le temps total d'exécution peut être réduit de 9% à 46% et l'amélioration en moyenne est de plus de 26% comparé à l'utilisation du modèle d'équilibrage de charge dynamique parallèle, qui ne prend pas en compte les dispositfs dynamiques et hétérogènes des systèmes distribués.

Nous arrivons enfin, après différentes approches (hypothèses restrictives ou bien encore applications parallèles de données) à des approches qui mèlent approche dymanique et statique ; ce qui nous permettra ensuite de faire le lien avec les approches uniquement statiques. Dans [40], l'utilisation efficace des systèmes parallèles à mémoire distribuée exige que la charge sur chaque processeur soit bien équilibrée. Dans les cas où la charge change de manière imprévisible pendant le calcul, une stratégie d'équilibrage de charge dynamique est nécessaire. De nombreux problèmes d'équilibrage de charge ont été étudiés, en particulier dans le contexte des applications sur maille non structurée. L'équilibrage de charge statique peut être approximé par un problème de partitionnement de graphes et beaucoup d'algorithmes efficaces ont été développés. Un progrès significatif a été également accompli dans le développement des algorithmes d'équilibrage de charge dynamiques. Cet article se penche sur l'histoire et la situation actuelle des deux classes d'algorithmes, avec une emphase particulière pour les applications sur maillage. Cependant les algorithmes fondamentaux, y compris ceux pour le partitionnement de graphes, sont suffisamment génériques pour être applicables à d'autres applications.

2.1.2 Approche statique

L'équilibrage de charge statique, quant à lui est basé sur la distribution de données parmi les processeurs participants, avant l'exécution de l'algorithme.

Les stratégies statiques sont plus spécifiques que les stratégies dynamiques mais elles se révèlent utiles si suffisamment de connaissance peut être injectée dans le processus décisionnel d'ordonnancement et d'application. En d'autres termes, si les caractéristiques de la plate-forme cible (vitesses des processeurs et capacités des liens) et de l'application cible (coût des calculs et des communications associés à chaque partition de données) sont connues avec suffisamment de précision, alors un excellent niveau de performance peut être atteint par le biais de stratégies statiques. Toutefois, des schémas sophistiqués de distribution de données sont indispensables pour atteindre ce niveau de performance. De nombreux auteurs se sont intéressés à l'équilibrage de charge statique dans le cadre de noyaux d'algèbre linéaire sur plates-formes hétérogènes. Ces algorithmes ont, entre autres, été étudiés dans [43, 4]. Les principales conclusions de ces articles s'appliquent pour trois types de problèmes :

- La distribution de tâches indépendantes sur un réseau unidimensionnel (linéaire) de processeurs hétérogènes est facile.
- La distribution de tâches indépendantes sur une grille bidimensionnelle de processeurs est difficile. Nous devons rechercher la meilleure distribution du travail

sur ladite grille pour chaque arrangement de processeurs, le nombre de tels arrangements étant exponentiel en la taille de la grille.
– Ne pas prendre en compte les contraintes géométriques induites par les deux dimensions de la grille conduit à des partitionnements irréguliers qui permettent un bon équilibrage de charge mais sont bien plus difficiles à mettre en œuvre.

Dans cette perspective, nous montrerons que la distribution de tâches indépendantes sur un réseau unidimensionnel de processeurs devient difficile lorsque l'on prend en plus compte des communications.

Dans la littérature, nous allons trouver des articles traitant strictement de l'équilibrage de charge mais aussi des articles traitant de l'équilibrage de charge combiné à d'autres problèmes. Par exemple, A. Pinar et C. Aykanat ont étudié dans [58] la décomposition unidimensionnelle des tableaux non-uniformes de charge de travail pour l'équilibrage de charge optimal. Le problème d'équilibrage de charge peut être modélisé comme un problème de partitionnement de « chaînes-sur-chaînes » (PCC) avec des poids de tâches positifs et non nuls ainsi que des arêtes sans poids entre les tâches successives. L'objectif du problème PCC est de trouver une suite de $P-1$ séparateurs pour diviser une chaîne de N tâches en P parties consécutives et manière à ce que la charge maximum sur les processeurs soit minimisée. Pour améliorer le temps d'exécution de leurs algorithmes, les auteurs utilisent une heuristique efficace comme étape de prétraitement pour trouver une borne supérieure de la charge maximum des processeurs. Ils fournissent aussi des pseudo-codes détaillés de leurs algorithmes de sorte que leurs résultats puissent être facilement reproduits. Par contre, A. Grimshaw et J. Weissman présentent dans [34] un cadre de travail qui automatise le partitionnement et le placement de calculs parallèles sur des systèmes hétérogènes. Ce partitionnement est réalisé lorsque l'état du système est connu. Trois problèmes sont traités : le choix du processeur, le placement de tâches et la répartition des données. La résolution de chacun de ces problèmes contribue à la réduction du temps d'exécution. En particulier, le choix du processeur détermine le meilleur découpage des données, le placement des tâches réduit le coût de communication et la répartition des données permet l'équilibrage de charge. Les auteurs commencent par décrire leur système ainsi que les suppositions qu'ils font à propos des communications et des processeurs. Ils discutent ensuite du placement des données, de la méthode de partionnement comportant trois phases d'exécution, et de l'implémentation en C++ qui montre comment les informations provenant des ressources et du programme sont utilisées.

De nombreux auteurs se sont aussi placés dans le cas d'un environnement distribué pour étudier les stratégies d'équilibrage de charge statiques. Tout d'abord, A. Heddaya et K. Park caractérisent et analysent dans [38] les conditions de communication d'un grand nombre d'applications qui tombent dans la sous-catégorie des problèmes de point fixe, résolvables par des méthodes itératives parallèles. Ceci a comme conséquence un ensemble d'interfaces et de dispositifs architecturaux suffisant pour l'exécution efficace des applications au-dessus d'un système distribué à grande échelle. Plus précisemment, ils proposent un lien direct entre l'application et la couche réseau. Ils se sont concentrés en particulier sur des méthodes itératives asynchrones qui ad-

mettent des communications non-bloquantes. Ceci augmente la réponse du système à la congestion du réseau, tout en améliorant l'exécution. Ils ont aussi formalisé et étendu leurs résultats aux systèmes distribués à mémoire partagée. De plus, ils récapitulent les résultats préliminaires d'un système prototype, montrant ainsi l'efficacité de leur modèle sur des calculs parallèles à grande échelle. Quant à S. Ichikawa et S. Yamashita, ils décrivent dans [41] un modèle d'équilibrage de charge statique pour des solveurs d'équations différentielles dans un environnement distribué. Bien qu'il y ait eu beaucoup de recherches sur l'équilibrage de charge statique, un environnement distribué est en terme de calculs une cible plus difficile du fait qu'il est composé de processeurs variés de caractéristiques différentes. Leur méthode considère à la fois les temps de calcul et de communication pour minimiser le temps total d'exécution avec partitionnement automatique des données et allocation de proceseurs. Ce problème est énoncé comme une optimisation combinatoire et résolu par la méthode de « branch-and-bound » jusqu'à 24 processeurs. Ils présentent également des algorithmes d'approximation qui donnent de bons résultats pour l'allocation et le partitionnement. La qualité de l'approximation est évaluée de manière quantitative en comparaison avec la solution optimale ou avec les bornes inférieures théoriques.

Nous pouvons aussi parler d'équilibrage de charge statique en théorie des jeux, notamment avec l'équilibre de Nash [35]. D. Grosu et A. T. Chronopoulos formulent le problème d'équilibrage de charge dans les systèmes distribués comme un jeu non coopératif parmi des utilisateurs, en utilisant l'équilibre de Nash (ici pour les stratégies pures). Il faut entendre par équilibre de Nash en stratégies pures, un concept de stabilité, une situation où aucun joueur n'a intérêt à dévier unilatéralement (individuellement) de sa stratégie ; c'est donc un ensemble d'actions tel que l'action de chaque joueur est une meilleure réponse aux actions choisies par les autres joueurs. Et donc, un équilibre de Nash pour le jeu d'équilibrage de charge de D. Grosu et A. T. Chronopoulos est un ensemble de stratégies avec la propriété qu'aucun utilisateur ne peut faire décoître son temps d'exécution en choisissant une stratégie d'équilibrage de charge différente étant données les stratégies d'équilibrage de charge des autres utilisateurs. Dans ce cas précis d'équilibrage de charge, il existe un unique équilibre de Nash. Le but des auteurs est alors de trouver un cadre de travail formel afin de caractériser les schémas d'allocation optimaux du point de vue utilisateur dans des systèmes distribués (chaque ordinateur du système étant modélisé comme une file d'attente M/M/1). L'équilibre de Nash permet d'obtenir une solution optimale pour l'utilisateur dans ce contexte. Les auteurs nous donnent une caractérisation de l'équilibre de Nash et un algorithme distribué pour le calculer. Afin de déterminer une solution pour leur jeu d'équilibrage de charge, ils considèrent une définition alternative de l'équilibre de Nash. L'équilibre de Nash peut être défini comme un ensemble de stratégies pour lequel la stratégie d'équilibrage de charge de chaque utilisateur est une meilleure réponse aux stratégies des autres utilisateurs. Cette définition donne donc une méthode pour déterminer la structure de l'équilibre de Nash pour le jeu de l'équilibrage de charge. Tout d'abord, ils déterminent les meilleures stratégies pour chaque joueur et trouvent un ensemble des stratégies. Le problème du calcul de la meilleure

stratégie de réponse d'un utilisateur se réduit au calcul de la stratégie optimale pour un sytème ayant un utilisateur et n ordinateurs. Le calcul de l'équilibre de Nash nécessite certaines coordinations entre les utilisateurs. Dans le cas présent, il est nécessaire que les utilisateurs obtiennent les informations relatives à la charge de chaque ordinateur. Une fois l'équilibre de Nash atteint, les utilisateurs continueront à utiliser les mêmes stratégies et le système restera équilibré. Cet équilibre sera maintenu jusqu'à ce qu'une nouvelle exécution de l'algorithme soit lancée. Les auteurs comparent ensuite les performances de leur algorithme avec trois algorithmes statiques existants (schéma proportionnel [15], schéma optimal global [46], et shéma optimal individuel [45]). Les principales conclusions qui résultent de ces comparaisons sont les avantages de leur algorithme d'équilibrage de charge : une structure distribuée, une faible complexité et une allocation optimale pour chaque utilisateur.

2.2 Redistribution

Les algorithmes de redistribution ont suscité une littérature abondante. Du côté théorique, dans le cadre de travail de la compilation de HPF [50], Kremer [51] a montré la NP-complétude du problème de redistribution. Ce résultat négatif montre que des algorithmes optimaux peuvent être conçus seulement pour des cas particuliers, tels que l'architecture d'anneau. Plusieurs algorithmes efficaces ont été conçus pour des anneaux [37, 53, 24], des arbres ou des hypercubes [68]. Notamment dans [24], E. Deelman et B. Szymanski présentent un algorithme d'équilibrage de charge dynamique. L'environnement de simulation est une grille 2D comportant des objets mobiles. Cette grille pourrait être découpée selon 2 dimensions mais cela pose des problèmes de rééquilibrage de charge. (Un avantage du partitionnement 2D serait d'avoir des coûts de communication faibles entre les frontières.) La grille est donc découpée selon une seule dimension, à savoir en tranches verticales ce qui permet d'équilibrer la charge plus facilement. Afin d'évaluer la charge, le nombre d'objets peut être compté et le nombre d'événements ayant eu lieu ou devant avoir lieu calculé. Les auteurs ont pris en compte les événements planifiés avec un poids plus ou moins grand : plus le poids est important, plus l'événement doit se produire rapidement. Tout le problème se résume donc à un rééquilibrage de charge sur un anneau de processeurs.

Un cas proche de la redistribution de données sur un anneau est la distribution cyclique des blocs de tableaux de données. Elle joue un rôle très important dans les bibliothèques scientifiques [9]. Dans la redistribution `CYCLIC(r)` sur p processeurs, des blocs de r éléments consécutifs d'un tableau sont distribués cycliquement, et le paramètre r est choisi de manière à optimiser la granularité, c'est-à-dire le ratio calcul-communication. Parce que cette granularité change d'un noyau à un autre, passer d'une distribution `CYCLIC(r)` sur p processeurs à une distribution `CYCLIC(s)` sur q processeurs est un procédé de redistribution très utilisé qui a été implémenté avec un algorithme de chenille dans ScaLAPACK [59]. Plusieurs articles, notamment [44, 66, 25, 57, 32, 14, 49], ont traité de diverses optimisations de ce procédé de

redistribution. Toujours dans cette optique, des outils automatiques de redistribution de données sont présentés dans [32] par J. Garcia, E. Ayguadé et J. Labarta. Leur cadre de travail est basé sur une nouvelle approche qui permet aux problèmes d'alignement, de distribution, et de redistribution d'être résolus ensemble en utilisant une représentation graphique simple. Comparé aux approches précédentes, leur algorithme combine distribution de données et redistribution dynamique avec des informations sur le parallélisme. Le graphe de parallélisme et de communication (CPG) est donc la structure qui contient les informations symboliques à propos du mouvement potentiel des données et du parallélisme inhérent du programme. Toutes ces informations sont pondérées en unités de temps et représentent le coût du déplacement de données ainsi que le coût de calcul. Le CPG est alors personnalisé pour une taille de problème et un système cible de données et est employé pour trouver un chemin de coût minimal dans le graphe en utilisant un solveur de programmation linéaire en nombre entier 0-1, ce qui garantie l'optimalité.

La redistribution de données permet aussi une grande adaptabilité au niveau logiciel. Par exemple, l'algorithme d'équilibrage de charge élastique conçu dans [56, 8] a mené à un logiciel de redistribution de données utilisé pour le traitement de requêtes [12] et l'analyse d'images médicales [61].

Un autre exemple dans lequel la redistribution de données est utile au niveau logiciel : dans [49], J. Knoop et E. Mehofer présentent une approche nouvelle et agressive afin d'éviter les ré-allocations inutiles et ayant pour but d'éliminer les changements de distribution partiellement redondants. Fondamentalement, cette approche évolue en prolongeant et en combinant deux algorithmes atteignant chacun ses propres résultats optimaux. La puissance et la flexibilité de cette nouvelle approche sont démontrées par les divers exemples, qui s'étendent des fragments typiques de HPF (High Performance Fortran) à de vrais programmes de la vie réelle. Les expériences soulignent son importance et montrent son efficacité sur différentes plates-formes et différents arrangements.

En dernier lieu, nous mentionnons brièvement trois applications dont l'exécution peut directement tirer bénéfice des stratégies de redistribution conçues dans le chapitre 4 de cette thèse. L'analyse des impulsions se propageant dans un milieu nonlinéaire réclame des fenêtres de calcul, et la redistribution doit se produire fréquemment pendant que le calcul progresse [10]. Un procédé à deux niveaux de redistribution est préconisé dans [52] pour l'amélioration de maillage. Une diffusion à multiniveaux est présentée dans [62, 63] pour des calculs irréguliers de grille et a été incorporée à la bibliothèque PARMETIS [47]. Naturellement, cette courte liste pourrait être fortement allongée.

Chapitre 3

Placement et équilibrage de charge

Kamitaz : *J'ai un petit problème avec mon avion 3D : je suis passé du pack d'accu 500 AR Ni-Cd sanyo 150 gr avec un bon équilibrage à un accu lipo 720 mAh 11,1V ; mon avion est extrêmement déséquilibré à l'arrière, que dois-je faire ?*
Buzz_l_éclair : *La résistance au crash d'un avion est inversement proportionnelle à la vitesse à laquelle il percute le mur !*
Forum « Modélisme »

3.1 Introduction

Nous nous intéressons à la mise en œuvre d'algorithmes itératifs sur des grappes hétérogènes. Ces algorithmes fonctionnent avec un volume important de données, qui sera réparti sur l'ensemble des processeurs. À chaque itération, des calculs indépendants seront effectués en parallèle et certaines communications auront lieu. Nous énonçons le problème comme suit : l'algorithme itératif fonctionne répétitivement sur une matrice rectangulaire de données qui est divisée en tranches verticales (ou horizontales) allouées aux processeurs. Il n'existe pas de raison *a priori* de réduire le partitionnement des données à une unique dimension. Cependant, un tel partitionnement est très naturel et nous montrerons que trouver l'optimal, dans ce cas, est déjà très difficile. Alors, à chaque étape de l'algorithme, les tranches sont mises à jour localement et les informations frontières sont échangées entre tranches consécutives. Cette contrainte géométrique (matrice découpée en tranches) implique que les processeurs soient organisés en anneau virtuel. Chaque processeur communiquera seulement deux fois, une fois avec son prédécesseur (virtuel) dans l'anneau et une fois avec son successeur.

Nous considérons une grappe totalement hétérogène, composée de processeurs de vitesses de calculs différentes, communicant par des liens de bandes passantes différentes. Du point de vue architectural, le problème se décompose en deux parties : (i) sélectionner les processeurs participant à la solution et décider de leur position dans l'anneau ; (ii) définir les chemins de communication pour aller d'un processeur à son successeur. Une difficulté majeure est que certains chemins partageront des liens physiques, les réseaux de communication de grappes hétérogènes n'étant pas totalement connectés. Si plusieurs chemins de communication partagent le même lien physique, nous décidons quelle fraction de bande passante sera attribuée à chaque route.

Une fois l'anneau et le routage décidés, il reste à déterminer le meilleur partitionnement des données. La qualité de la solution finale dépend d'un grand nombre de paramètres de l'application ainsi que de l'architecture, et il faut s'attendre à ce que le problème d'optimisation soit difficile à résoudre.

Pour évaluer l'impact du partage de la bande passante des liens, nous travaillons aussi avec la version simple du problème où nous voyons le réseau comme un graphe complet : entre chaque paire de nœuds, le routage est fixé (plus court chemin en terme de bande passante), et la bande passante est définie par le lien le plus lent dans le chemin routé. Ce modèle simplifié n'est pas très réaliste car aucun partage de lien n'est pris en compte, mais il mènera à un anneau solution qui pourra être comparé à celui obtenu en tenant compte du partage des liens, fournissant de ce fait une manière commode d'évaluer l'impact des différentes hypothèses faites sur les communications.

3.2 Modèles

Dans cette section, nous présentons tout d'abord les spécificités communes des deux problèmes d'optimisation que nous allons étudier : SLICERING et SHAREDRING. Nous établissons ensuite les caractéristiques de SLICERING dans la section 3.2.2 ainsi que les caractéristiques de SHAREDRING dans la section 3.2.3. Et nous discutons enfin dans la section 3.2.4 plusieurs variantes de notre modèle.

3.2.1 Caractéristiques communes

Coûts de calcul. La plate-forme de calcul ciblée est représentée par un graphe orienté $G = (P, E)$. Chaque nœud P_i du graphe, $1 \leqslant i \leqslant |P| = p$, représente une ressource de calcul, et est pondéré par son temps de cycle w_i : P_i nécessite w_i unités de temps pour effectuer une tâche unitaire.

Coûts des communications. Les arêtes du graphe représentent les liens de communication et sont étiquetées avec les bandes passantes disponibles. Si $e \in E$ est un lien orienté de P_i à P_j, notons b_e la bande passante du lien. Nous aurons besoin de

D_c/b_e unités de temps pour transférer un message de taille D_c de P_i à P_j en utilisant le lien e.

Paramètres de l'application : calculs. Soit D_w la taille totale du travail qui doit être accompli à chaque itération de l'algorithme. Le processeur P_i effectuera une quantité de travail $\alpha_i.D_w$ de ce travail, où $\alpha_i \geqslant 0$ pour $1 \leqslant i \leqslant p$ et $\sum_{i=1}^{p} \alpha_i = 1$. Notons que $\alpha_j = 0$, pour un certain j, signifie que le processeur P_j ne participe pas au calcul. En effet, il n'y a aucune raison *a priori* pour que toutes les ressources soient utilisées, surtout lorsque le travail n'est pas très important : les communications supplémentaires encourues en ajoutant plus de processeurs peuvent ralentir l'ensemble du processus en dépit de l'augmentation de la vitesse de traitement cumulée.

Paramètres de l'application : communications. Les processeurs sont organisés le long d'un anneau (qui n'est pas encore déterminé). Après avoir mis à jour ses données de taille $\alpha_i D_w$, chaque processeur P_i envoie un message de longueur D_c fixée (typiquement, la taille des données frontières) à son successeur. Pour illustrer la relation entre D_w et D_c, nous pouvons voir la matrice de données comme un rectangle composé de D_w colonnes de hauteur D_c, ainsi une seule colonne est échangée entre paire de processeurs consécutifs dans l'anneau (le paramètre D_c peut être représenté comme un volume fixe de communication).

Soit succ(i) et pred(i) le successeur et le prédécesseur de P_i dans l'anneau virtuel. Le temps nécessaire pour transférer un message de taille D_c de P_i à P_j est $D_c.c_{i,j}$ où $c_{i,j}$ est la capacité des liens de communication entre P_i à P_j. $c_{i,j}$ varie selon que l'on suppose que les liens sont partagés ou non.

Fonction objective. Le coût total d'une seule étape de l'algorithme itératif est le maximum, parmi tous les processeurs impliqués, du temps passé à calculer et à communiquer :

$$T_{\text{step}} = \max_{1 \leqslant i \leqslant p} \mathbb{I}\{i\}[\alpha_i.D_w.w_i + D_c.(c_{i,\text{pred}(i)} + c_{i,\text{succ}(i)})] \tag{3.1}$$

où $\mathbb{I}\{i\}[x] = x$ si P_i participe au calcul et 0 sinon. En résumé, le but est de déterminer la meilleure méthode pour sélectionner q processeurs parmi les p disponibles, de leur assigner des charges de travail, de les arranger le long d'un anneau et de partager la bande passante du réseau, de telle sorte que le temps total d'exécution d'une itération soit minimal.

3.2.2 Spécificités pour SLICERING

La plate-forme ciblée est modélisée par un graphe complet afin de simplifier le problème d'optimisation SHAREDRING (cf. section 3.4.5 pour un exemple dans lequel un graphe complet est construit à partir d'une plate-forme réelle en utilisant l'algorithme de plus court chemin en terme de bande passante.)

Coûts des communications. Le temps nécessaire pour transférer un message de taille D_c de P_i à P_j est égal à $D_c.c_{i,j}$, où $c_{i,j}$ est la capacité du lien e, de P_i à P_j, c'est-à-dire, l'inverse de sa bande passante b_e.

Paramètres de l'application : communications. Le processeur P_i a besoin de $D_c.c_{i,\text{succ}(i)}$ unités de temps pour envoyer un message de taille D_c à son successeur, ainsi que de $D_c.c_{i,\text{pred}(i)}$ unités de temps pour envoyer un message de même taille à son prédécesseur.

3.2.3 Spécificités pour SHAREDRING

Les communications et le routage sont plus difficiles à traiter dans le cas du problème d'optimisation SHAREDRING qui est plus général.

Coûts des communications. Nous aurons besoin de D_c/b_e unités de temps pour transférer un message de taille D_c de P_i à P_j en utilisant le lien e. Lorsque plusieurs messages partagent le même lien, chacun reçoit une fraction de la bande passante disponible. Si deux messages partagent le même lien e et si le premier message utilise deux tiers de la bande passante, c'est-à-dire $2b_e/3$, alors le second message ne pourra utiliser qu'au plus $b_e/3$. Les fractions de bandes passantes qui sont allouées aux messages peuvent être déterminées par l'utilisateur à la seule condition que la somme de ces fractions ne dépasse pas la bande passante totale du lien. En pratique, un protocole tel que celui décrit dans [48] nous offre une telle liberté pour la stratégie de routage.

Routage. Supposons que nous pouvons décider comment les messages sont routés d'un processeur à un autre et que nous voulons router un message de taille D_c de P_i à P_j, en passant par k arêtes $e_1, e_2, \ldots, e_k$. Pour chaque arête e_m, le message aura une fraction f_m de la bande passante b_{e_m}. La vitesse globale de communication le long du chemin sera limitée par la plus petite bande passante disponible : nous avons besoin de D_c/b unités de temps pour router le message, où $b = \min_{1 \leqslant m \leqslant k} f_m$: c'est comme si nous avions un lien direct dédié au routage de ce message mais de bande passante réduite b.

Paramètres de l'application : communications. Il existe un chemin de communication $\mathcal{S}_i$ ($\mathcal{S}$ pour « successeur ») de P_i à $P_{\text{succ}(i)}$ dans le réseau : soit $s_{i,m}$ la fraction de la bande passante b_{e_m} du lien physique e_m qui a été allouée pour le chemin $\mathcal{S}_i$. Évidemment, si un lien e_r n'est pas utilisé dans le chemin, alors $s_{i,r} = 0$. Soit $c_{i,\text{succ}(i)} = \frac{1}{\min_{e_m \in \mathcal{S}_i} s_{i,m}}$: alors P_i a besoin de $D_c.c_{i,\text{succ}(i)}$ unités de temps pour envoyer un message de taille D_c à son successeur $P_{\text{succ}(i)}$. De même, nous définissons le chemin de communication $\mathcal{P}_i$ ($\mathcal{P}$ pour « prédécesseur ») de P_i à $P_{\text{pred}(i)}$ dans le réseau ; $p_{i,m}$ est la fraction de la bande passante b_{e_m} du lien physique e_m qui a été allouée pour le

chemin $\mathcal{P}_i$, et $c_{i,\text{pred}(i)} = \frac{1}{\min_{e_m \in \mathcal{P}_i} p_{i,m}}$. Alors P_i a besoin de $D_c.c_{i,\text{pred}(i)}$ unités de temps pour envoyer un message de taille D_c à son prédécesseur $P_{\text{pred}(i)}$.

3.2.4 Variations sur le modèle

Nous discutons ici de plusieurs variantes de notre précédent cadre de travail :

Surcoûts d'initialisation. La motivation à utiliser un modèle simple de coût linéaire, plutôt qu'un modèle affine impliquant des surcoûts, pour les communications et les calculs, est la suivante : seules des applications à grande échelle sont susceptibles d'être déployées sur les plates-formes hétérogènes. Les surcoûts d'initialisation peuvent donc être négligés. Quoi qu'il en soit, la plupart des résultats présentés ici se prolongent à un modèle affine.

Liens bidirectionnels. Il est facile de modéliser un lien bidirectionnel entre deux processeurs donnés P_i et P_j : indépendamment de leur orientation (de P_i à P_j ou l'inverse), toutes les communications utilisant ce lien auront une partie de la bande passante, de manière à ce que la bande passante totale disponible sur ce lien ne soit pas dépassée. En d'autres termes, nous allouons une fraction f_{path} de la bande passante à chaque chemin de communication passant par le lien de bande passante b sans prendre en compte son orientation et nous établissons la contrainte $\sum f_{\text{path}} \leqslant b$; la somme se prolonge à tous les chemins utilisant le lien, indépendamment de leur orientation. En fait, les liens unidirectionnels et bidirectionnels peuvent simultanément exister dans le réseau, et il est facile de les modéliser tous les deux.

Liens multiples. De la même manière, les liens multiples entre une paire de processeurs donnés peuvent être facilement pris en compte : il suffit de modéliser G comme un graphe multiple plutôt que comme un graphe simple.

Liens centraux. Les liens de « backbones » peuvent fournir à plusieurs communications le même taux de bande passante b : pour modéliser un tel lien, nous allouons la même fraction $f_{\text{path}} = b$ à chaque chemin passant par le lien, indépendamment de leur nombre. En d'autres termes, nous remplaçons la contrainte $\sum f_{\text{path}} \leqslant b$ par $f_{\text{path}} \leqslant b$ pour chaque chemin utilisant le lien.

Pour conclure cette section, nous précisons que ce cadre n'est pas limité aux algorithmes itératifs. En fait, notre approche s'applique aux problèmes où des calculs indépendants sont répartis sur des ressources hétérogènes disposées le long d'un anneau, et sont intercalés avec des communications entre des processeurs adjacents. L'hypothèse principale est que les communications se produisent seulement entre les processeurs adjacents, et que leur volume est indépendant de leur charge de travail relative.

Il existe d'autres modèles architecturaux qui seraient intéressants à étudier. Nous employons un modèle où chaque processeur envoie séquentiellement des messages à ses deux voisins, et nous supposons implicitement des réceptions asynchrones. En

particulier, nous pourrions supposer que chaque processeur est capable d'envoyer les deux messages en parallèle, ce qui mènerait à un opérateur *Max* au lieu d'une somme dans les surcoûts de communication de l'équation 3.1.

3.3 Le problème d'optimisation SLICERING

3.3.1 Plan d'attaque

Dans cette section, nous établissons formellement le problème d'optimisation à résoudre. Comme il a déjà été dit, la plate-forme ciblée est modélisée par un graphe complet G. Chaque nœud P_i du graphe, $1 \leqslant i \leqslant |P| = p$ représente une ressource de calcul, et est pondéré par son temps de cycle relatif w_i : P_i a besoin de w_i unités de temps pour effectuer une tâche unitaire. Les arêtes sont étiquetées avec les coûts de communication : le temps nécessaire pour transférer un message de taille D_c de P_i à P_j est égal à $D_c.c_{i,j}$ où $c_{i,j}$ est la capacité du lien de communication entre P_i et P_j, c'est-à-dire l'inverse de sa bande passante. Le but de notre problème est de minimiser le maximum des coûts de calcul et de communication entre des processeurs voisins. Le problème d'optimisation s'écrit de la manière suivante :

Définition 3.1 (SLICERING(p,w_i,$c_{i,j}$,D_w,D_c)). *Soient p processeurs de temps de cycle w_i et $p(p-1)$ liens de communication de capacité $c_{i,j}$, soient D_w la charge totale de travail et D_c le volume de communication à chaque étape, déterminons*

$$T_{step} = \min_{\substack{1 \leqslant q \leqslant p, \\ \sigma \in \Theta_{q,p}, \\ \sum_{i=1}^{q} \alpha_{\sigma(i)} = 1}} \left\{ \max_{1 \leqslant i \leqslant q} \left(\alpha_{\sigma(i)}.D_w.w_{\sigma(i)} + D_c.(c_{\sigma(i),\sigma(i-1 \bmod q)} + c_{\sigma(i),\sigma(i+1 \bmod q)}) \right) \right\} \quad (3.2)$$

Dans l'équation 3.2, $\Theta_{q,p}$ est l'ensemble des fonctions $\sigma : [1..q] \to [1..p]$ qui indexent les q processeurs sélectionnés, pour toutes les valeurs de q comprises entre 1 et p. De cette équation, il n'est pas clair, que tous les processeurs soient impliqués ou non. Ce sera le cas seulement si le rapport $\frac{D_w}{D_c}$ est assez grand. Étant donné les valeurs des paramètres D_c et D_w, il existe une borne supérieure en ce qui concerne les q processeurs participants : au-dessus de cette valeur, le coût relatif des communications devient trop important devant la charge moyenne de calcul (ce que nous mettons en évidence sur les figures 3.13 à 3.16). Dans tous les cas, après avoir décidé combien et quels processeurs utiliser, nous devons encore décider comment les arranger en anneau. Extraire le « meilleur » anneau du graphe d'interconnexion semble être un problème combinatoire difficile. Avant d'évaluer ce résultat (ce que nous montrons à la section 3.3.4), nous traitons la situation beaucoup plus facile lorsque le réseau est homogène (cf. la section 3.3.2).

3.3.2 Plate-forme homogène

La solution du problème d'optimisation, c'est-à-dire, minimiser l'équation (3.2), est facile quand tous les temps de communication sont égaux. Ceci correspond à un réseau homogène où chaque paire de processeurs peut communiquer à la même vitesse, par exemple via un bus ou un « backbone » Ethernet.

Supposons que $c_{i,j} = c$ pour tout i et j, où c est une constante. Il y a seulement deux cas à considérer : (i) seul le processeur le plus rapide est actif ; (ii) tous les processeurs participent. En effet, dès qu'une communication simple se produit, nous pouvons en avoir plusieurs pour le même coût, et le meilleur est de diviser la charge de calcul parmi toutes les ressources. Dans le premier cas, nous obtenons que $T_{\text{step}} = D_w.w_{\min}$, où $w_{\min}$ est le plus petit temps de cycle. Dans le second cas, la charge est plus équilibrée lorsque le temps d'exécution est le même pour tous les processeurs : sinon, enlever une petite fraction de travail au processeur qui a le plus grand temps d'exécution et la donner au processeur qui finit le plus tôt ferait diminuer le temps de calcul maximum. Cela entraîne $\alpha_i.w_i$ = Constante pour tout i avec $\sum_{i=1}^{p} \alpha_i = 1$. Nous obtenons que $T_{\text{step}} = D_w.w_{\text{cumul}} + 2D_c.c$, où $w_{\text{cumul}} = \frac{1}{\sum_{i=1}^{p} \frac{1}{w_i}}$. Ces résultats sont résumés par :

Proposition 3.1. *La solution optimale de* SLICERING*(p,w_i,c,D_w,D_c) est*

$$T_{step} = \min\{D_w.w_{\min}, D_w.w_{cumul} + 2D_c.c\}$$

où $w_{\min} = \min_{1 \leqslant i \leqslant p} w_i$ *et* $w_{cumul} = \frac{1}{\sum_{i=1}^{p} \frac{1}{w_i}}$.

Si la plate-forme est donnée, il existe un seuil, qui est dépendant de l'application, pour décider si seulement le processeur le plus rapide, par opposition à toutes les ressources, doit être impliqué. D_c étant donné, le processeur le plus rapide fera tout le travail pour des petites valeurs de D_w, à savoir $D_w \leqslant D_c.\frac{2c}{w_{\min} - w_{\text{cumul}}}$. Sinon, pour de plus grandes valeurs de D_w, tous les processeurs doivent être inclus.

3.3.3 Plate-forme hétérogène

3.3.3.1 Programmation linéaire en entiers : introduction

Lorsque le réseau est hétérogène, nous nous retrouvons face à une situation complexe : déterminer le nombre de processeurs qui doivent prendre part au calcul est déjà une question difficile. Dans cette section, nous exprimons la solution du problème d'optimisation SLICERING en termes de Programmation Linéaire en Entiers (PLE). Bien sûr, la complexité de cette approche est exponentielle dans le pire cas, mais elle fournira des idées utiles pour concevoir des heuristiques peu coûteuses. Nous commençons par le cas où tous les processeurs sont impliqués dans la solution optimale. Nous étendons cette approche au cas général ensuite.

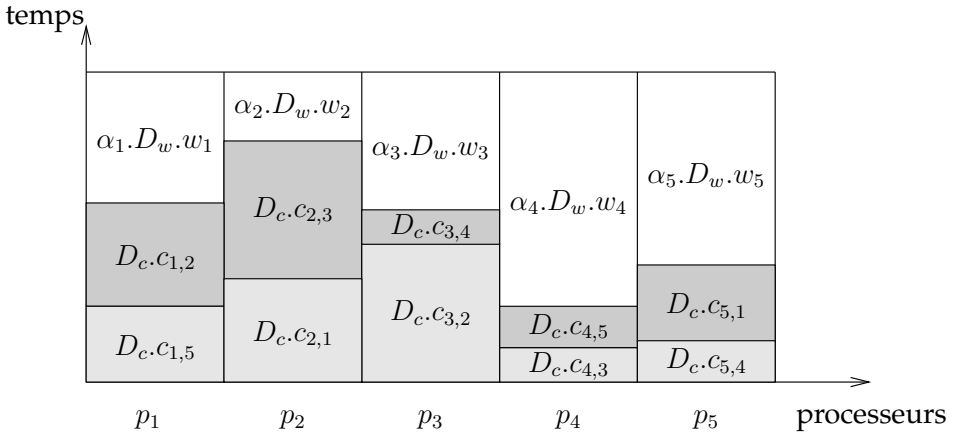


FIG. 3.1 – Résumé des temps de calcul et de communication avec $p = 5$ processeurs.

3.3.3.2 Solution avec implication de tous les processeurs

Supposons tout d'abord que tous les processeurs sont impliqués dans la solution optimale. Tous les processeurs requièrent le même temps pour calculer et communiquer : sinon, nous pourrions sensiblement diminuer la charge de calcul du processeur le plus lent et donner la charge supplémentaire à un autre processeur. Ainsi (cf figure 3.1 pour une illustration), nous obtenons :

$$T_{\text{step}} = \alpha_i.D_w.w_i + D_c.(c_{i,i-1} + c_{i,i+1}) \tag{3.3}$$

pour tout i (les indices dans les coûts de communication sont pris modulo p). Puisque $\sum_{i=1}^{p} \alpha_i = 1$, nous obtenons que $\sum_{i=1}^{p} \frac{T_{\text{step}} - D_c.(c_{i,i-1}+c_{i,i+1})}{D_w.w_i} = 1$. Posons $w_{\text{cumul}} = \frac{1}{\sum_{i=1}^{p} \frac{1}{w_i}}$, nous obtenons ainsi :

$$\frac{T_{\text{step}}}{D_w.w_{\text{cumul}}} = 1 + \frac{D_c}{D_w} \sum_{i=1}^{p} \frac{c_{i,i-1} + c_{i,i+1}}{w_i} \tag{3.4}$$

T_{step} sera alors minimal lorsque $\sum_{i=1}^{p} \frac{c_{i,i-1}+c_{i,i+1}}{w_i}$ sera minimal. Cela sera atteint pour l'anneau qui correspond au plus court cycle Hamiltonien dans le graphe $G = (P, E)$, où chaque arête $e_{i,j}$ est donnée par le poids $d_{i,j} = \frac{c_{i,j}+c_{j,i}}{w_i}$. Une fois que nous avons ce chemin, nous obtenons T_{step} à partir de l'équation 3.4, et nous déterminons alors la charge α_i de chaque processeur en utilisant l'équation 3.3.

Afin de résumer, nous avons le résultat suivant :

Proposition 3.2. *Lorsque tous les processeurs sont impliqués, trouver la solution optimale revient à résoudre le problème du voyageur de commerce dans un graphe pondéré (P, E, d), où $d_{i,j} = \frac{c_{i,j}}{w_i} + \frac{c_{j,i}}{w_j}$.*

Bien sûr, nous ne nous attendons pas à une solution en temps polynomial pour ce résultat, puisque le problème de décision associé au problème du voyageur de commerce est NP-complet [33] (la distance d ne satisfait pas ici l'inégalité triangulaire,

il n'existe pas d'algorithme en temps polynomial [19]). Mais cette équivalence nous donne deux directions :

- Pour des plates-formes de petite taille, la solution optimale peut être calculée en utilisant un programme linéaire en entier, qui retourne la solution optimale du problème de voyageur de commerce.
- Pour de très grandes plates-formes, nous pouvons utiliser des heuristiques qui approchent la solution du problème du voyageur de commerce en temps polynomial, comme l'heuristique de Lin-Kernighan [55, 39].

Dans la suite, nous rappelons brièvement la formulation classique du problème du voyageur de commerce comme un problème de programmation linéaire en entier (PLE). Cela permettra de résoudre notre problème lorsque tous les processeurs seront inclus. Dans la sous-section 3.3.3.3 , nous étendrons la formulation PLE au cas où seulement une partie des processeurs sont impliqués dans la solution optimale.

Considérons le graphe complet pondéré $G = (P, E, d)$, où $|P| = p$, et supposons que nous commençons le tour, c'est-à-dire l'anneau de processeurs, avec le processeur P_1. Soient $x_{i,j}$ les variables entières telles que $x_{i,j} = 1$ lorsque P_j est le processeur suivant P_i dans l'anneau, et $x_{i,j} = 0$ sinon. Puisqu'un processeur exactement précède P_j dans l'anneau, nous avons $\sum_{i=1}^{p} x_{i,j} = 1$ pour chaque j. De manière similaire, nous avons $\sum_{j=1}^{p} x_{i,j} = 1$ pour chaque i. Le coût d'un tour peut être exprimé de la façon suivante : $\sum_{i=1}^{p} \sum_{j=1}^{p} d_{i,j}.x_{i,j}$.

Mais ces équations ne sont pas suffisantes : nous devons exclure le cas de deux (ou plus) sous-tours disjoints. Dans cette optique, nous introduisons $p-1$ nouvelles variables entières $u_2, u_3, \ldots, u_p$ avec $u_i \geqslant 0$ et $(p-1)(p-2)$ nouvelles contraintes comme suit :

$$u_i - u_j + p.x_{i,j} \leqslant p-1 \text{ pour } 2 \leqslant i, j \leqslant p,\ i \neq j.$$

Intuitivement, u_i représente la position dans le tour de P_i, et les contraintes assurent que le tour n'est pas découpé en sous-tours. En effet, suivons la preuve de [20]. Supposons tout d'abord que nous avons un cycle Hamiltonien (« commençant » à P_1) : nous allons prouver que le problème PLE possède une solution. Soit u_i la position dans le chemin de P_i (excluant P_1, et comptant à partir de 0). (Par exemple, pour l'instance avec $p = 5$ et le tour $P_1 \to P_4 \to P_2 \to P_3 \to P_5 \to P_1$, alors nous avons $u_4 = 0$, $u_2 = 1$, $u_3 = 2$, et $u_5 = 3$.) Revenons au cas général : nous avons toujours $0 \leqslant u_i \leqslant p-2$ pour $i \geqslant 2$. Par conséquent, si $x_{i,j} = 0$, $u_i - u_j + p.x_{i,j} \leqslant p-2$ si $i \neq j$, et l'inégalité tient. Ensuite, si $x_{i,j} = 1$, P_j est visité immédiatement après P_i, alors $u_j = u_i + 1$, et $u_i - u_j + p.x_{i,j} = p-1$, et l'inégalité tient encore. Réciproquement, supposons que nous avons une solution au problème PLE, et que le tour est découpé en au moins deux sous-tours. Alors, il existe un sous-tour de $r \leqslant p-1$ processeurs qui n'inclut pas P_1. Ajoutant les r équations pour les r valeurs non nulles de $x_{i,j}$ de ce sous-tour, cela nous mène à $r.p \leqslant r.(p-1)$ (toutes les variables u_i apparaissent deux fois et s'annulent), une contradiction. En conclusion, nous sommes amenée à la formulation PLE suivante :

Formulation en programmation linéaire en entiers du voyageur de commerce
MINIMISER $\sum_{i=1}^{p}\sum_{j=1}^{p} d_{i,j}.x_{i,j}$,
AVEC LES CONTRAINTES SUIVANTES :

$$\begin{cases} (1)\ \sum_{j=1}^{p} x_{i,j} = 1 & 1 \leqslant i \leqslant p \\ (2)\ \sum_{i=1}^{p} x_{i,j} = 1 & 1 \leqslant j \leqslant p \\ (3)\ x_{i,j} \in \{0,1\} & 1 \leqslant i,j \leqslant p \\ (4)\ u_i - u_j + p.x_{i,j} \leqslant p-1 & 2 \leqslant i,j \leqslant p, i \neq j \\ (5)\ u_i \text{ entier}, u_i \geqslant 0 & 2 \leqslant i \leqslant p \end{cases}$$

Proposition 3.3. *Lorsque tous les processeurs sont impliqués, trouver la solution optimale revient à résoudre le programme linéaire précédent.*

3.3.3.3 Solution dans le cas général

Comment étendre la formulation PLE au cas général ? Pour chaque valeur possible de q, $1 \leqslant q \leqslant p$, nous énoncerons un problème PLE donnant la solution optimale avec exactement q processeurs participant. En prenant la solution la plus petite parmi les p valeurs retournées par ces problèmes PLE, nous obtiendrons la solution optimale.

Pour une valeur donnée de q, $1 \leqslant q \leqslant p$, nous utilisons une technique similaire à celle de la section 3.3.3.2, mais nous avons besoin de variables supplémentaires.

Formulation en programmation linéaire en entier pour un anneau de taille q
MINIMISER T,
AVEC LES CONTRAINTES SUIVANTES :

$$\begin{cases} (1)\ x_{i,j} \in \{0,1\} & 1 \leqslant i,j \leqslant p \\ (2)\ \sum_{i=1}^{p} x_{i,j} \leqslant 1 & 1 \leqslant j \leqslant p \\ (3)\ \sum_{i=1}^{p}\sum_{j=1}^{p} x_{i,j} = q & \\ (4)\ \sum_{i=1}^{p} x_{i,j} = \sum_{i=1}^{p} x_{j,i} & 1 \leqslant j \leqslant p \\ (5)\ \sum_{i=1}^{p} \alpha_i = 1 & \\ (6)\ \alpha_i \leqslant \sum_{j=1}^{p} x_{i,j} & 1 \leqslant i \leqslant p \\ (7)\ \alpha_i.w_i + \frac{D_c}{D_w}\sum_{j=1}^{p}(x_{i,j}c_{i,j} + x_{j,i}c_{j,i}) \leqslant T & 1 \leqslant i \leqslant p \\ (8)\ \sum_{i=1}^{p} y_i = 1 & \\ (9)\ -p.y_i - p.y_j + u_i - u_j + q.x_{i,j} \leqslant q-1 & 1 \leqslant i,j \leqslant p, i \neq j \\ (10)\ y_i \in \{0,1\} & 1 \leqslant i \leqslant p \\ (11)\ u_i \text{ entier}, u_i \geqslant 0 & 1 \leqslant i \leqslant p \end{cases}$$

Comme précédemment, l'idée intuitive est que $x_{i,j} = 1$ si et seulement si P_j est le successeur immédiat de P_i dans l'anneau de taille q. Les contraintes (2) et (4) établissent que le degré entrant de chaque nœud est le même que son degré sortant, et sera égal à 0 ou 1. La contrainte (3) assure que l'anneau est bien composé de q processeurs. La contrainte (5) assure que la quantité de travail allouée à l'ensemble des

processeurs est bien égale à D_w. La contrainte (6) assure quant à elle que le travail est bien réparti entre les processeurs faisant partie de l'anneau solution. La contrainte (7) est la traduction du fait qu'on cherche à minimiser le temps des phases de calcul et de communication dans l'anneau. À partir des contraintes (8) et (10), nous voyons qu'un seul y_i sera non nul, et qu'il représentera « l'origine » de l'anneau de taille q. Supposons que la valeur non nulle est y_1. Pour $i = 1$ et pour toute valeur de j, la contrainte (9) sera satisfaite à cause du terme $-p.y_1$. Si ni i ni j n'est égal à l'origine P_1, (9) se réduit à la contrainte (1) du problème du voyageur de commerce, et assure que l'anneau de taille q n'est pas découpé en sous-anneaux. Dans la solution, $u_i = 0$ pour l'origine et pour les processeurs ne participant pas à l'anneau solution, et u_i est la position après l'origine (numéroté de 0 à $q - 2$) du nœud P_i dans l'anneau.

Nous résumons ces résultats de la manière suivante :

Proposition 3.4. *Le problème d'optimisation* SLICERING *peut être résolu en calculant la solution de p programmes linéaires en entier, où p est le nombre total de ressources.*

3.3.4 Complexité

Le problème de décision associé au problème d'optimisation SLICERING est :

Définition 3.2 (SLICERINGDEC(p,w_i,$c_{i,j}$,D_w,D_c,K)). *Étant donnés p processeurs de temps de cycle w_i et $p(p-1)$ liens de communication de capacité $c_{i,j}$, étant donnés la charge de travail totale D_w et D_c le volume de communication à chaque étape, et étant donnée une borne de temps K, est-il possible de trouver un entier $q \leqslant p$, une application $\sigma : [1..q] \to [1..p]$, et des nombres rationnels positifs α_i avec $\sum_{i=1}^{q} \alpha_{\sigma(i)} = 1$, tels que*

$$T_{step} = \max_{1 \leqslant i \leqslant q} \left\{ \alpha_{\sigma(i)}.D_w.w_{\sigma(i)} + D_c.(c_{\sigma(i),\sigma(i-1 \bmod q)} + c_{\sigma(i),\sigma(i+1 \bmod q)}) \right\} \leqslant K ?$$

Le résultat suivant énonce la difficulté intrinsèque du problème :

Théorème 3.1. SLICERINGDEC(p,w_i,$c_{i,j}$,D_w,D_c,K) *est NP-complet.*

Démonstration. Clairement, SLICERINGDEC appartient à NP. Pour prouver qu'il est NP-complet, nous utilisons une réduction de HAMPATH, le problème de cycle Hamiltonien qui est NP-complet [33]. Considérons une instance arbitraire $\mathcal{I}_1$ de HAMPATH : étant donné un graphe $G_h = (V_h, E_h)$, existe-t-il un cycle Hamiltonien dans G_h, c'est-à-dire un cycle passant une et une seule fois par tous les sommets de G ?

Nous construisons l'instance suivante $\mathcal{I}_2$ de SLICERINGDEC : soit $p = |V_h|$ (supposons $p \geqslant 2$ sans perte de généralité) et définissons un graphe complet d'interconnection $G = (P, E)$, dont le coût des arêtes est donné par

$$c_e = \begin{cases} \varepsilon & \text{si } e \in E_h \\ 2 & \text{sinon} \end{cases}$$

où $0 < \varepsilon < \frac{1}{2}$ est une constante. Soit $D_w = D_c = 1$ et $w_i = p$ pour $1 \leqslant i \leqslant p$. Clairement, $\mathcal{I}_2$ peut être construit en temps polynomial à partir de $\mathcal{I}_1$. Finalement, soit $K = 1 + 2\varepsilon$.

Supposons premièrement que $\mathcal{I}_1$ a une solution, c'est-à-dire que G_h contient un cycle Hamiltonien. Nous utilisons les arêtes de ce chemin pour construire l'anneau. Tous les processeurs sont inclus, et soit $\alpha_i = 1/p$ pour $1 \leqslant i \leqslant p$. Le temps d'exécution et le temps de communication sont les mêmes pour tous les processeurs et nous obtenons que $T_{\text{step}} = \frac{1}{p} \cdot p + 2\varepsilon = K$, par conséquent une solution pour $\mathcal{I}_2$.

Supposons maintenant que $\mathcal{I}_2$ a une solution. Si un seul processeur participait à cette solution, nous devrions avoir $T_{\text{step}} = 1.p \geqslant 2 > K$, ce qui est une contradiction. Par conséquent, il a q processeurs, avec $q \geqslant 2$, qui participent à la solution. Si l'anneau utilisait une arête de communication qui n'appartenait pas à G_h, alors le coût de communication de cette arête devrait être de 2 et $T_{\text{step}} \geqslant D_c.2 = 2 > K$, ce qui nous donne encore une contradiction. Il reste à prouver que nous utilisons tous les p processeurs dans la solution. Sinon, si $q < p$, une charge de travail serait au moins égale à $\frac{1}{q}.D_w.p > 1$, ce qui impliquerait que $T_{\text{step}} > K$. En conclusion, $q = p$, et les arêtes de l'anneau solution permettent de construire un cycle Hamiltonien dans G_h, et de ce fait, une solution à $\mathcal{I}_1$. ■

3.3.5 Heuristiques

Après les résultats théoriques précédents, nous adoptons une approche plus pratique dans cette section. Notre but est d'obtenir des heuristiques en temps polynomial afin de résoudre le problème d'optimisation SLICERING.

Après avoir exprimé le problème en termes d'une collection de programmes linéaires en entier, nous sommes capable de calculer la solution optimale avec des outils tels que PIP [30, 29] ou LP_SOLVE [5] (au moins pour des tailles raisonnables des plates-formes de calcul). Nous comparons cette solution optimale avec celles retournées par les deux heuristiques en temps polynomial, une qui approxime le problème du voyageur de commerce (mais qui retourne une solution seulement si tous les processeurs sont impliqués dans le calcul), et l'autre, une heuristique gloutonne qui agrandit itérativement l'anneau solution.

3.3.5.1 Heuristique basée sur le principe du voyageur de commerce

La situation où tous les processeurs sont impliqués dans la solution optimale est très importante en pratique. En effet, seules les très grandes applications sont déployées sur des plates-formes hétérogènes distribuées. Et quand D_w est suffisamment grand, nous obtenons de l'équation 3.5 que tous les processeurs seront impliqués.

D'après la section 3.3.3.2, nous savons que la solution optimale, lorsque tous les processeurs sont impliqués, correspond au plus court cycle Hamiltonien dans le graphe (P, E, d), avec $d_{i,j} = \frac{c_{i,j}}{w_i} + \frac{c_{j,i}}{w_j}$. Nous utilisons l'heuristique de Lin-Kernighan [55,

39], afin d'approximer le plus court chemin. Par construction, l'heuristique basée sur le problème du voyageur de commerce retourne toujours une solution où tous les processeurs sont impliqués. Évidemment, si la solution optimale nécessite peu de processeurs, l'heuristique basée sur le problème du voyageur de commerce donne une solution de mauvaise qualité.

3.3.5.2 Heuristique gloutonne

L'heuristique gloutonne commence par sélectionner le processeur le plus rapide. Puis, elle insère itérativement un nouveau nœud dans l'anneau solution en cours. Supposons que nous avons déjà sélectionné un anneau de r processeurs. Pour chaque processeur P_i restant, nous cherchons où l'insérer dans l'anneau courant. Donc, pour chaque paire consécutive de processeurs (P_j, P_k) dans l'anneau et pour chaque processeur P_i restant, nous calculons la nouvelle valeur de T_{step}. Nous retenons alors le processeur et la paire qui minimisent la valeur de T_{step} et nous stockons cette valeur. Cette étape de l'heuristique a une complexité proportionnelle à $(p-r).r$.

Ensuite, nous faisons grossir l'anneau jusqu'à avoir p processeurs et nous retournons la valeur minimale obtenue pour T_{step}. La complexité totale est $\sum_{r=1}^{p}(p-r)r = O(p^3)$. Notons qu'il est important d'essayer toutes les valeurs de r, parce que T_{step} ne varie pas de façon monotone en fonction de r.

3.3.6 Simulations

3.3.6.1 Description de la plate-forme

La figure 3.2 montre la plate-forme de Lyon, composée de 14 unités de calcul et 3 routeurs. Sur cette figure les nœuds 0 à 13, en forme de cercle, sont les processeurs, alors que les nœuds 14 à 16, en forme de losange, sont les routeurs. Les temps de cycle des processeurs (en secondes par mégaflop) sont rassemblés dans le tableau 3.1. De façon similaire, la plate-forme de Strasbourg (figure 3.3) est composée de 13 unités de calcul et de 6 routeurs. Les temps de cycle des processeurs sont rassemblés dans le tableau 3.2. Il est à noter que les caractéristiques des deux plates-formes ont été mesurées afin d'obtenir des simulations basées sur la « réalité ».

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}	P_{16}
0,0206	0,0206	0,0206	0,0206	0,0291	0,0206	0,0087	0,0206	0,0206	0,0206	0,0206	0,0206	0,0291	0,0451	0	0	0

TAB. 3.1 – Temps de cycle des processeurs (en secondes par mégaflop) pour la plate-forme de Lyon.

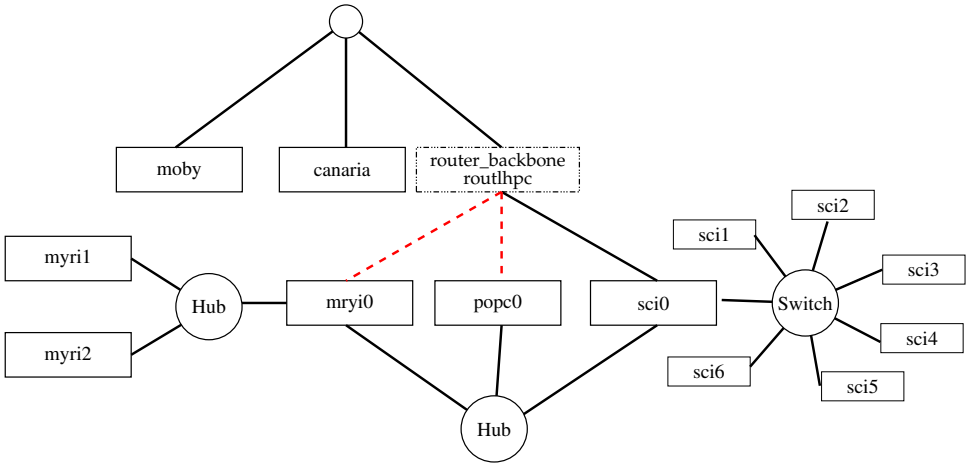


FIG. 3.2 – Topologie de la plate-forme de Lyon.

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}	P_{16}	P_{17}	P_{18}
0,0087	0,0072	0,0087	0,0131	0,016	0,0058	0,0087	0,0262	0,0102	0,0131	0,0072	0,0058	0,0072	0	0	0	0	0	0

TAB. 3.2 – Temps de cycle des processeurs (en secondes par mégaflop) pour la plate-forme de Strasbourg.

3.3.6.2 Résultats

Pour les deux topologies, nous comparons l'heuristique gloutonne et la solution optimale obtenue par les logiciels de programmation linéaire en entier, lorsque cette solution est disponible. Comme LP_SOLVE ne réussit pas à calculer le résultat lorsqu'il y a plus de 5 processeurs dans la solution, nous reportons seulement les résultats obtenu par le logiciel PIP. Les tableaux 3.3 et 3.4 montrent la différence entre l'heuristique gloutonne et la solution optimale obtenue avec PIP sur les plates-formes de Lyon et Strasbourg. Les nombres dans les tableaux représentent le coût d'un chemin de longueur q sur la plate-forme, c'est-à-dire la valeur de la fonction objectif de la programmation linéaire en entier de la section 3.3.3.3 (multipliée par un facteur 6000 puisque PIP a besoin de matrice d'entiers).

PIP est capable de calculer la solution optimale pour toutes les valeurs sur la plate-forme de Strasbourg, mais échoue entre 9 et 13 processeurs sur la plate-forme de Lyon (notez que nous avons utilisé une machine avec deux gigaoctets de RAM !) Lorsque tous les processeurs sont impliqués, nous avons essayé l'heuristique LKH : pour les deux plates-formes, elle retourne les résultats optimaux. Les conclusions qui se dessinent de ces expériences sont les suivantes :

- l'heuristique gloutonne est rapide et plus efficace, à 11.2% de l'optimal pour la plate-forme de Lyon, et 6.8% pour la plate-forme de Strasbourg.
- l'heuristique LKH est vraiment très efficace mais son application est limitée au cas où toutes les ressources sont impliquées.
- les logiciels de programmation linéaire en entiers échouent rapidement sur le

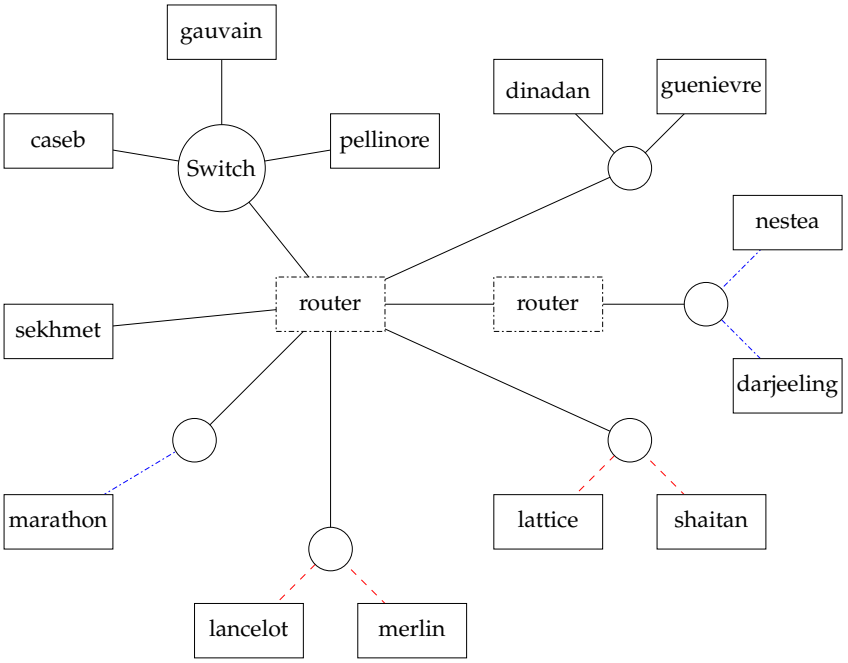


FIG. 3.3 – Topologie de la plate-forme de Strasbourg.

calcul de la solution optimale.

Heuristiques \ Processeurs	3	4	5	6	7	8	9	10	11	12	13	14
Gloutonne	1202	556	1152	906	2240	3238	4236	5234	6232	7230	8228	10077
PIP	878	556	1128	906	2075	3071	hors mémoire	hors mémoire	hors mémoire	hors mémoire	hors mémoire	9059

TAB. 3.3 – Comparaison entre l'heuristique gloutonne et PIP pour la plate-forme de Lyon.

Sur les figures 3.4 et 3.5, nous avons tracé le nombre p_{opt} de processeurs dans la solution optimale en fonction du ratio D_w/D_c. Comme prévu, lorsque le ratio croît (ce qui signifie plus de calcul par rapport aux communications), de plus en plus de processeurs sont utilisés dans la solution optimale et la valeur de p_{opt} augmente. Comme les liens de la plate-forme de Lyon ont des capacités similaires, la valeur de p_{opt} passe de 1 (solution séquentielle) à 14 (tous les processeurs participent) alors que l'heuristique gloutonne trouve une solution à 6 processeurs entre temps. Le saut de 1 à 14 est facile à expliquer : du fait que les communications coûtent plus ou moins le même prix, même s'il y a peu de travail, autant partager la charge entre tous les processeurs.

Le réseau d'interconnexion de la plate-forme de Strasbourg est plus hétérogène et donc, la valeur de p_{opt} passe de 1 (solution séquentielle), à 10, puis 11 et enfin 13 (tous les processeurs sont inclus), tout comme l'heuristique gloutonne qui suit le même schéma.

Heuristiques \ Processeurs	3	4	5	6	7	8	9	10	11	12	13
Gloutonne	1520	2112	3144	3736	4958	5668	7353	8505	10195	12490	15759
PIP	1517	2109	3141	3733	4955	5660	7348	8500	10188	12235	14757

TAB. 3.4 – Comparaison entre l'heuristique gloutonne et PIP pour la plate-forme de Strasbourg.

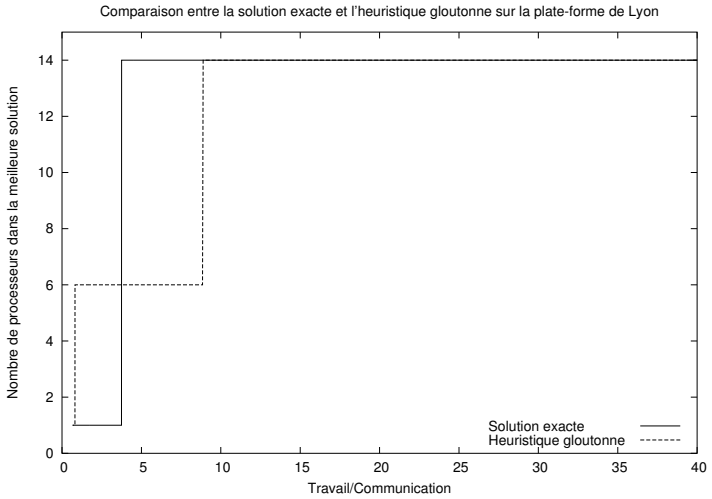


FIG. 3.4 – Nombre optimal de processeurs pour la plate-forme de Lyon.

3.3.7 Conclusion

La principale limitation de la programmation sur plates-formes hétérogènes provient de la difficulté à équilibrer la charge. Les données et les calculs ne sont pas forcément distribués à l'ensemble des processeurs. Minimiser les surcoûts de communication devient alors une tâche ardue.

Le résultat majeur de cette section est la NP-complétude du problème d'optimisation SLICERING. Plus que la preuve, le résultat lui-même est intéressant, puisqu'il fournit une nouvelle preuve de la difficulté intrinsèque de la conception d'algorithmes hétérogènes. Une autre avancée importante est la conception d'une heuristique efficace qui fournit une ligne de conduite pragmatique au concepteur de calculs scientifiques itératifs.

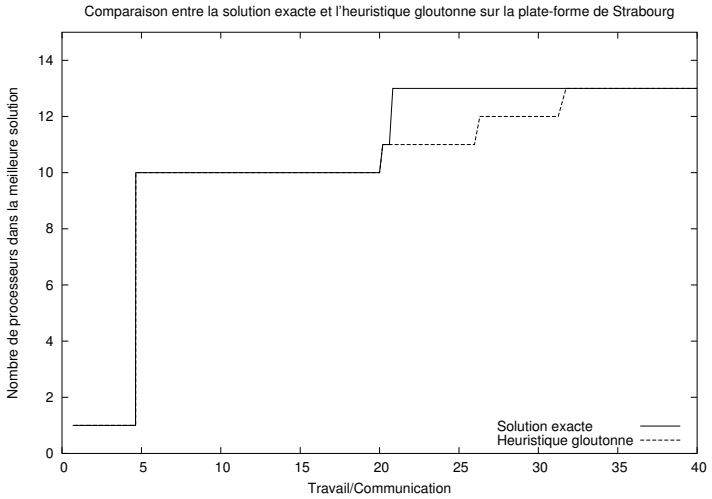


FIG. 3.5 – Nombre optimal de processeurs pour la plate-forme de Strasbourg.

3.4 Le problème d'optimisation SHAREDRING

3.4.1 Plan d'attaque

Définition 3.3 (SHAREDRING(p,w_i,E,b_{e_m},D_w,D_c)). *Étant donnés p processeurs P_i de temps de cycle w_i et $|E|$ liens de communication e_m de bande passante b_{e_m}, étant donnés D_w la charge de travail totale et D_c le volume de communication à chaque étape, minimiser*

$$T_{step} = \min_{1 \leqslant q \leqslant p} \quad \min_{\substack{\sigma \in \Theta_{q,p} \\ \sum_{i=1}^{q} \alpha_{\sigma(i)} = 1}} \quad \max_{1 \leqslant i \leqslant q} \left(\alpha_{\sigma(i)} . D_w . w_{\sigma(i)} + D_c . \left(c_{\sigma(i),\sigma(i-1 \bmod q)} + c_{\sigma(i),\sigma(i+1 \bmod q)} \right) \right) \tag{3.5}$$

Dans l'équation 3.5, $\Theta_{q,p}$ symbolise l'ensemble des fonctions injectives $\sigma : [1..q] \rightarrow [1..p]$, lesquelles indexent les q processeurs sélectionnés pour l'anneau, pour tout q compris entre 1 et p. Pour chaque anneau candidat représenté par une telle fonction σ, il existe des contraintes cachées par l'introduction des quantités $c_{\sigma(i),\sigma(i-1 \bmod q)}$ et $c_{\sigma(i),\sigma(i+1 \bmod q)}$: il existe $2q$ chemins de communication, le chemin $\mathcal{S}_i$ de $P_{\sigma(i)}$ à son successeur $P_{\text{succ}(\sigma(i))} = P_{\sigma(i+1 \bmod q)}$ et le chemin $\mathcal{P}_i$ de $P_{\sigma(i)}$ à son prédécesseur $P_{\text{pred}(\sigma(i))} = P_{\sigma(i-1 \bmod q)}$, pour tout $1 \leqslant i \leqslant q$. Pour chaque lien e_m dans le réseau interconnecté, soit $s_{\sigma(i),m}$ (resp. $p_{\sigma(i),m}$) la fraction de la bande passante b_{e_m} qui est allouée au chemin $\mathcal{S}_{\sigma(i)}$ (resp. $\mathcal{P}_{\sigma(i)}$). Nous obtenons les équations suivantes :

$$\begin{cases} s_{\sigma(i),m} \geqslant 0, \ p_{\sigma(i),m} \geqslant 0 & 1 \leqslant i \leqslant q, 1 \leqslant m \leqslant E \\ c_{\sigma(i),\text{succ}(\sigma(i))} = \frac{1}{\min_{e_m \in \mathcal{S}_{\sigma(i)}} s_{\sigma(i),m}} & 1 \leqslant i \leqslant q \\ c_{\sigma(i),\text{pred}(\sigma(i))} = \frac{1}{\min_{e_m \in \mathcal{P}_{\sigma(i)}} p_{\sigma(i),m}} & 1 \leqslant i \leqslant q \\ \sum_{i=1}^{q} (s_{\sigma(i),m} + p_{\sigma(i),m}) \leqslant b_{e_m} & 1 \leqslant m \leqslant E \end{cases}$$

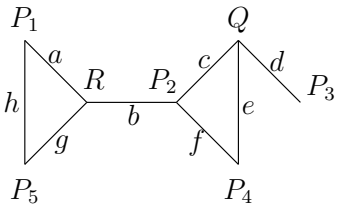


FIG. 3.6 – Un cluster de petite taille.

La dernière équation vient du fait que la bande passante du lien e_m ne doit pas être dépassée. Puisque chaque chemin de communication $\mathcal{S}_{\sigma(i)}$ ou $\mathcal{P}_{\sigma(i)}$ n'impliquera que peu d'arêtes, la plupart des quantités $s_{\sigma(i),m}$ et $p_{\sigma(i),m}$ seront égales à zéro. En fait, nous avons écrit $e_m \in \mathcal{S}_{\sigma(i)}$ si l'arête e_m est utilisée dans le chemin $\mathcal{S}_{\sigma(i)}$, c'est-à-dire si $s_{i,m}$ ne vaut pas zéro (de même $e_m \in \mathcal{P}_{\sigma(i)}$ si $p_{i,m}$ ne vaut pas zéro).

Comme pour notre problème d'optimisation SLICERING, plus le ratio $\frac{W}{H}$ sera grand, plus le nombre de processeurs augmentera dans la solution optimale. Mais en plus, nous devons encore décider comment organiser les processeurs en anneau, construire les routes de communication, assigner les fractions de bandes passantes et finalement allouer les données.

3.4.2 Exemple

Considérons le réseau hétérogène représenté à la figure 3.6. Il contient 7 processeurs et 8 liens de communication bidirectionnels. Pour simplifier, nous avons étiqueté les processeurs P_1 à P_5 dans l'ordre dans lequel ils apparaissent dans l'anneau solution de 5 processeurs que nous construisons, en laissant de côté les processeurs Q et R. Les liens sont étiquetés par des lettres de a à h au lieu d'indices ; nous utilisons b_x pour noter la bande passante du lien x.

Pour le chemin de P_1 à P_2, nous choisissons d'utiliser les liens a et b et, donc, le chemin de communication de P_1 à son successeur P_2 est $\mathcal{S}_1 = \{a, b\}$. Par contre, pour le chemin de P_2 à P_1, nous utiliserons plutôt les liens b, g, et h et, donc, le chemin de communication de P_2 à son prédécesseur est $\mathcal{P}_2 = \{b, g, h\}$. Voici une liste complète des chemins que nous avons décidé d'utiliser (à noter que de nombreux autres choix peuvent être faits) :

- De P_1 : vers P_2, $\mathcal{S}_1 = \{a, b\}$ et vers P_5, $\mathcal{P}_1 = \{h\}$
- De P_2 : vers P_3, $\mathcal{S}_2 = \{c, d\}$ et vers P_1, $\mathcal{P}_2 = \{b, g, h\}$
- De P_3 : vers P_4, $\mathcal{S}_3 = \{d, e\}$ et vers P_2, $\mathcal{P}_3 = \{d, e, f\}$
- De P_4 : vers P_5, $\mathcal{S}_4 = \{f, b, g\}$ et vers P_3, $\mathcal{P}_4 = \{e, d\}$
- De P_5 : vers P_1, $\mathcal{S}_5 = \{h\}$ et vers P_4, $\mathcal{P}_5 = \{g, b, f\}$

Nous définissons ensuite les coûts des chemins. Pour P_1, parce que $\mathcal{S}_1 = \{a, b\}$, nous obtenons $c_{1,2} = \frac{1}{\min(s_{1,a}, s_{1,b})}$; et parce que $\mathcal{P}_1 = \{h\}$, nous obtenons $c_{1,5} = \frac{1}{p_{1,h}}$. Nous

procédons de la même manière de P_2 à P_5, et ainsi de suite. Nous aboutissons alors à la liste de toutes les contraintes qui doivent être satisfaites :

Lien a **:** $s_{1,a} \leqslant b_a$
Lien b **:** $s_{1,b} + s_{4,b} + p_{2,b} + p_{5,b} \leqslant b_b$
Lien c **:** $s_{2,c} \leqslant b_c$
Lien d **:** $s_{2,d} + s_{3,d} + p_{3,d} + p_{4,d} \leqslant b_d$
Lien e **:** $s_{3,e} + p_{3,e} + p_{4,e} \leqslant b_e$
Lien f **:** $s_{4,f} + p_{3,f} + p_{5,f} \leqslant b_f$
Lien g **:** $s_{4,g} + p_{2,g} + p_{5,g} \leqslant b_g$
Lien h **:** $s_{5,h} + p_{1,h} + p_{2,h} \leqslant b_h$

Maintenant que nous avons toutes ces contraintes, nous pouvons (essayer de) calculer les α_i, $s_{i,j}$ et $p_{i,j}$ qui minimisent la fonction objective T_{step}. L'équation 3.6 explicite le système complet d'(in)équations qui est quadratique en les inconnues α_i, $s_{i,j}$ et $p_{i,j}$ [1]. La série d'inéquations sous la forme $s_{\sigma(i),m}.c_{\sigma(i),succ(\sigma(i))} \geqslant 1$ est la traduction de la série d'équations $c_{\sigma(i),succ(\sigma(i))} = \frac{1}{\min_{em \in \mathcal{S}_{\sigma(i)}} s_{\sigma(i),m}}$. Cela nous permet donc de montrer de manière explicite que le programme est quadratique du fait de la multiplication des inconnues $s_{\sigma(i),m}$ et $c_{\sigma(i),succ(\sigma(i))}$ entre elles.

Minimiser $\max_{1 \leqslant i \leqslant 5} (\alpha_i.D_w.w_i + D_c.(c_{i,i-1} + c_{i,i+1}))$ avec les contraintes suivantes :

$$\begin{cases} \sum_{i=1}^{5} \alpha_i = 1 & & \\ s_{1,a} \leqslant b_a & s_{1,b} + s_{4,b} + p_{2,b} + p_{5,b} \leqslant b_b & s_{2,c} \leqslant b_c \\ s_{2,d} + s_{3,d} + p_{3,d} + p_{4,d} \leqslant b_d & s_{3,e} + p_{3,e} + p_{4,e} \leqslant b_e & s_{4,f} + p_{3,f} + p_{5,f} \leqslant b_f \\ s_{4,g} + p_{2,g} + p_{5,g} \leqslant b_g & s_{5,h} + p_{1,h} + p_{2,h} \leqslant b_h & \\ s_{1,a}.c_{1,2} \geqslant 1 & s_{1,b}.c_{1,2} \geqslant 1 & p_{1,h}.c_{1,5} \geqslant 1 \\ s_{2,c}.c_{2,3} \geqslant 1 & s_{2,d}.c_{2,3} \geqslant 1 & p_{2,b}.c_{2,1} \geqslant 1 \\ p_{2,g}.c_{2,1} \geqslant 1 & p_{2,h}.c_{2,1} \geqslant 1 & s_{3,d}.c_{3,4} \geqslant 1 \\ s_{3,e}.c_{3,4} \geqslant 1 & p_{3,d}.c_{3,2} \geqslant 1 & p_{3,e}.c_{3,2} \geqslant 1 \\ p_{3,f}.c_{3,2} \geqslant 1 & s_{4,f}.c_{4,5} \geqslant 1 & s_{4,b}.c_{4,5} \geqslant 1 \\ s_{4,g}.c_{4,5} \geqslant 1 & p_{4,e}.c_{4,3} \geqslant 1 & p_{4,d}.c_{4,3} \geqslant 1 \\ s_{5,h}.c_{5,1} \geqslant 1 & p_{5,g}.c_{5,4} \geqslant 1 & p_{5,b}.c_{5,4} \geqslant 1 \\ p_{5,f}.c_{5,4} \geqslant 1 & & \end{cases} \tag{3.6}$$

Pour construire l'équation 3.6, nous avons utilisé des chemins de communication arbitraires, et il en existe beaucoup d'autres à essayer. De même pour les autres anneaux à construire, que ce soit avec les mêmes processeurs ou avec des processeurs différents. Le nombre de processeurs q peut aussi varier. Sans aucune surprise, nous montrons maintenant que le problème de décision associé au problème d'optimisation SHAREDRING est NP-complet.

[1] Nous n'avons pas exprimé dans l'équation 3.6 les inéquations qui établissent que toutes les inconnues sont strictement positives.

3.4.3 Complexité

Le problème de décision associé au problème d'optimisation SHAREDRING est le suivant :

Définition 3.4 (SHAREDRINGDEC(p,w_i,E,b_{e_m},D_w,D_c,K)). *Étant donnés p processeurs P_i de temps de cycle w_i et E liens de communication e_m de bande passante b_{e_m}, étant donnés la charge totale de travail D_w et le volume de communication D_c à chaque étape, et étant donnée une borne de temps K, est-il possible de trouver un sous-ensemble de $q \leqslant p$ processeurs, une application $\sigma : [1..q] \to [1..p]$, $2q$ chemins de communication $\mathcal{S}_i$ et $\mathcal{P}_i$ tels que aucune bande passante totale de lien ne soit pas dépassée, et des nombres rationnels strictement positifs α_i avec $\sum_{i=1}^{q} \alpha_{\sigma(i)} = 1$, tels que $T_{step} \leqslant K$, où T_{step} est donné par l'équation 3.5 ?*

Le résultat suivant établit la difficulté intrinsèque du problème SHAREDRING :

Théorème 3.2. SHAREDRINGDEC(p,w_i,E,b_{e_m},D_w,D_c,K) *est NP-complet.*

Démonstration. Clairement, SHAREDRINGDEC appartient à NP. Pour prouver qu'il est NP-complet, nous utilisons une nouvelle fois une réduction de HAMPATH, le problème de chemin hamiltonien qui est NP-complet [33]. Considérons une instance arbitraire $\mathcal{I}_1$ de HAMPATH : étant donné un graphe $G_h = (V_h, E_h)$, existe-t-il un cycle Hamiltonien dans G_h, c'est-à-dire un cycle passant une et une seule fois par tous les sommets de G ?

Nous construisons l'instance suivante $\mathcal{I}_2$ de SHAREDRINGDEC : soit $p = |V_h|$ (supposons $p \geqslant 2$ sans perte de généralité) et définissons un graphe complet d'interconnection $G = (P, E)$. Dans G, toutes les arêtes sont unidirectionnelles : en particulier, de chaque arête de E_h, nous obtenons deux arêtes de E. Les bandes passantes des arêtes dans E sont données par :

$$b_e = \begin{cases} 1/\varepsilon & \text{si } e \text{ est obtenue à partir de } e \in E_h \\ 1/2 & \text{sinon} \end{cases}$$

où $0 < \varepsilon < \frac{1}{2}$ est une constante. Soient $D_w = D_c = 1$ et $w_i = p$ pour $1 \leqslant i \leqslant p$. Clairement, $\mathcal{I}_2$ peut être construit en temps polynomial à partir de $\mathcal{I}_1$. Finalement, soit $K = 1 + 2\varepsilon$.

Supposons premièrement que $\mathcal{I}_1$ a une solution, c'est-à-dire que G_h contient un chemin Hamiltonien. Nous utilisons les arêtes de ce chemin pour construire l'anneau. Tous les processeurs sont inclus, et soit $\alpha_i = 1/p$ pour $1 \leqslant i \leqslant p$. À partir de maintenant, les indices sont pris modulo p. Nous ré-indexons les processeurs et les arêtes ; le chemin Hamiltonien est alors $e_1, e_2, \ldots, e_p$ où e_i connecte P_i à P_{i+1} ; de cette construction de G, il existe un chemin inverse, c'est-à-dire les arêtes e_{p+i} de P_{i+1} à P_i, vont dans la direction opposée des e_i. Pour $1 \leqslant i \leqslant p$, $\mathcal{S}_i$ se réduit à e_i, soient $s_{i,i+1} = 1/\varepsilon$ et $s_{i,m} = 0$ pour $m \neq i$. De façon similaire, $\mathcal{P}_i$ se réduit à e_{p+i-1} (sauf pour $\mathcal{P}_1$ qui se réduit à e_{2p}) ; soient $p_{i,p+i-1} = 1/\varepsilon$ et $p_{i,m} = 0$ pour $m \neq p+i-1$ (sauf pour $i = 1$: $p_{i,2p} = 1/\varepsilon$ et $p_{1,m} = 0$ pour $m \neq 2p$). Chaque arête e_i est utilisée une seule fois, et sa bande passante

totale $1/\varepsilon$ n'est pas dépassée. Le temps d'exécution et le temps de communication sont les mêmes pour tous les processeurs et égaux à $T_{\text{step}} = \frac{1}{p} \cdot p + 2\varepsilon = K$, par conséquent, nous obtenons une solution pour $\mathcal{I}_2$.

Supposons maintenant que $\mathcal{I}_2$ a une solution. Si un seul processeur participait à cette solution, nous devrions avoir $T_{\text{step}} = 1.p \geqslant 2 > K$, ce qui est une contradiction. Par conséquent, il a q processeurs, avec $q \geqslant 2$, qui participent à la solution. Si l'anneau utilisait une arête de communication qui n'appartenait pas à G_h, alors son coût devrait être au moins de 2 et $T_{\text{step}} \geqslant D_c.2 = 2 > K$, ce qui nous donne encore une contradiction. Il reste à prouver que nous utilisons tous les processeurs p dans la solution. Car sinon, si $q < p$, une charge de calcul serait au moins égale à $\frac{1}{q}.D_w.p > 1$, ce qui impliquerait que $T_{\text{step}} > K$. En conclusion, $q = p$, et toutes les arêtes de l'anneau solution permettent de trouver un chemin Hamiltonien dans G_h, et de ce fait, une solution à $\mathcal{I}_1$. ■

3.4.4 Heuristiques

Le problème d'optimisation SHAREDRING étant NP-complet, nous ne pouvons pas le résoudre directement ; nous allons donc le résoudre grâce à une heuristique. De plus, ce problème étant très complexe, nous allons procéder dans notre heuristique en plusieurs étapes.

Nous décrivons dans cette section une heuristique en temps polynomial pour construire un anneau solution et résoudre le problème d'optimisation SHAREDRING. Nous décrivons l'heuristique en trois étapes : (i) l'algorithme glouton utilisé pour construire un anneau solution ; (ii) la stratégie mise en œuvre pour assigner des fractions de bande passante pendant la construction ; et (iii) des optimisations finales.

3.4.4.1 Construction de l'anneau

Le principe de base de notre heuristique pour le problème d'optimisation SHARED-RING est le même que pour l'heuristique décrite en section 3.3.5.2.

Nous considérons un anneau solution impliquant q processeurs, numérotés de P_1 à P_q. Nous utiliserons l'équation 3.4 comme base pour un algorithme glouton générant itérativement un anneau solution. L'algorithme glouton commence par sélectionner la meilleure paire de processeurs. Ensuite, il inclut itérativement un nouveau processeur dans l'anneau solution actuel. Supposons que nous avons déjà sélectionné un anneau de r processeurs. Pour chaque processeur restant P_i, nous cherchons où l'insérer dans l'anneau actuel.

Nous avons donc besoin de calculer le coût d'insertion du processeur P_k entre les processeurs P_i et P_j, en d'autres termes calculer les nouveaux coûts $c_{k,j}$ (chemin de P_k à son successeur P_j), $c_{j,k}$ (le chemin inverse), $c_{k,i}$ (chemin de P_k à son prédécesseur P_i), et $c_{i,k}$ (chemin inverse). Pour cela, nous connaissons une heuristique qui construit les

chemins de communication et alloue des fractions de bande passante (les détails de l'allocation de bande passante sont en section 3.4.4.2). Une fois que nous avons calculé ces coûts, nous calculons la nouvelle valeur de T_{step} comme suit :

- Nous mettons à jour w_{cumul} en ajoutant le nouveau processeur P_k, ce qui a pour effet de diminuer w_{cumul}.
- Dans la somme $\sum_{s=1}^{r} \frac{c_{\sigma(s),\sigma(s-1)}+c_{\sigma(s),\sigma(s+1)}}{w_{\sigma(s)}}$, nous supprimons les deux termes correspondant aux deux chemins entre P_i et P_j (par hypothèse, il existe un s tel que $i = \sigma(s)$ et $j = \sigma(s+1)$), et nous insérons les nouveaux termes $\frac{c_{k,j}+c_{k,i}}{w_k}$, $\frac{c_{j,k}}{w_j}$ et $\frac{c_{i,k}}{w_i}$.

L'étape de l'heuristique que nous avons décrit précédemment a une complexité proportionnelle à $(p-r).r$ fois le coût pour calculer quatre chemins de communication. Pour finir, nous agrandissons l'anneau jusqu'à avoir p processeurs. Nous retournons alors la valeur minimale obtenue pour T_{step}. La complexité totale est $\sum_{r=1}^{p}(p-r)rC = O(p^3)C$, où C est le coût de calcul de quatre chemins dans le réseau. Notons une fois de plus qu'il est important de tester toutes les valeurs de r, car T_{step} peut ne pas varier de façon monotone avec r.

3.4.4.2 Allocation de bande passante

Dans cette section, nous supposons que nous avons déjà un anneau de r processeurs, une paire (P_i, P_j) de processeurs successifs dans l'anneau et un nouveau processeur P_k à insérer entre P_i et P_j. Parallèlement à l'anneau, nous avons construit $2r$ chemins de communication et une certaine fraction des bandes passantes initiales ont été allouées à chacun de ces chemins. Pour construire les quatre nouveaux chemins impliquant P_k, nous raisonnons sur le graphe $G = (V, E, b)$ où chaque arête est étiquetée par la bande passante disponible restante : maintenant $b(e_m)$ n'est plus la bande passante initiale de l'arête e_m, mais ce qui a été laissé par les $2r$ chemins. La première chose à faire est de ré-injecter dans le réseau les fractions de bande passante utilisées par les deux chemins de communication entre P_i et P_j. Nous utilisons un algorithme simple de calcul du plus court chemin (en termes de bande passante) pour déterminer les quatre chemins, de P_k à P_i et P_j et vice-versa. Il y a ici une subtilité, car ces quatre chemins peuvent partager certains liens. La stratégie que nous utilisons est la suivante :

- Nous calculons indépendamment quatre chemins de bande passante maximale, à l'aide d'un algorithme standard de calcul du plus court chemin [19] sur G.
- Si tous les chemins sont disjoints, alors la totalité de la bande passante disponible leur est attribuée.
- Sinon, si certains chemins partagent des liens, nous ne changeons pas les chemins, mais nous employons une méthode (analytique) brutale pour calculer les fractions de bande passante minimisant l'équation 3.4 devant être allouées à chaque chemin, et nous mettons à jour les coûts des chemins en conséquence.
- Nous ré-injectons un peu de bande passante utilisée précédemment par les chemins de communication afin de ne pas pénaliser certains processeurs.

Maintenant que nous avons les chemins et leurs coûts, nous calculons la nouvelle valeur de T_{step} comme expliqué précédemment. Le coût C de calcul des quatre chemins dans le réseau est $O(p + E)$.

3.4.4.3 Optimisations

Un moyen concis de décrire l'heuristique est le suivant : nous construisons un anneau de façon gloutonne en fractionnant les bandes passantes afin d'insérer de nouveaux processeurs. Pour diminuer le coût de l'heuristique, nous ne recalculons jamais les fractions de bande passante ayant été assignées aux précédents chemins de communication. Quand l'heuristique se termine, nous avons un anneau de q processeurs, $2q$ chemins de communication, $2q$ fractions de bande passante, $2q$ coûts de communication pour ces chemins, q charges de travail, et une valeur réaliste de T_{step}. Nous avons mis en œuvre deux variantes destinées à affiner cette solution. L'idée des deux variantes est de tout conserver, sauf les fractions de bande passante (et donc les charges de travail), et de recalculer celles-ci chaque fois que l'on a inséré un nouveau processeur dans l'anneau. En d'autres termes, une fois que nous avons sélectionné le processeur et la paire de processeurs minimisant le coût d'insertion dans l'anneau, nous effectuons l'insertion et nous recalculons toutes les fractions de bande passante et les charges de travail. Nous gardons l'anneau (les processeurs ainsi que l'ordre dans lequel ils apparaissent) et les chemins de communication en tant que tels. Puisque nous connaissons les $2q$ chemins, nous pouvons ré-évaluer les fractions de bande passante, en utilisant une approche globale :

Méthode 1 : Max-min fairness. C'est l'algorithme traditionnel de partage de bande passante [7], conçu pour maximiser la bande passante minimum allouée à un chemin.

Méthode 2 : résolution quadratique à l'aide du logiciel KINSOL [65]. Une fois obtenu un anneau ainsi que tous les chemins de communication, le programme pour minimiser T_{step} est quadratique en les inconnues α_i, $s_{i,j}$ et $p_{i,j}$ (même justification que pour l'exemple de la section 3.4.2). Nous utilisons la bibliothèque KINSOL pour le résoudre.

3.4.5 Simulations

3.4.5.1 Générateur de topologie

Afin d'évaluer les heuristiques écrites pour SHAREDRING, nous avons fait des simulations avec deux plates-formes générées par Tiers [13, 26]. Ce générateur produit des graphes ayant trois niveaux de hiérarchie référencés comme LAN, MAN et WAN : les LAN (« Local Area Network ») représentent un ensemble d'ordinateurs reliés entre eux dans une petite aire géographique par un réseau, souvent à l'aide d'une même technologie ; les MAN (« Metropolitan Area Network ») interconnectent plusieurs

LAN géographiquement proches à des débits importants. Ainsi un MAN permet à deux noeuds distants de communiquer comme si ils faisaient partie d'un même réseau local ; un WAN (« Wide Area Network ») interconnecte plusieurs LANs à travers de grandes distances géographiques.

3.4.5.2 Description des plates-formes

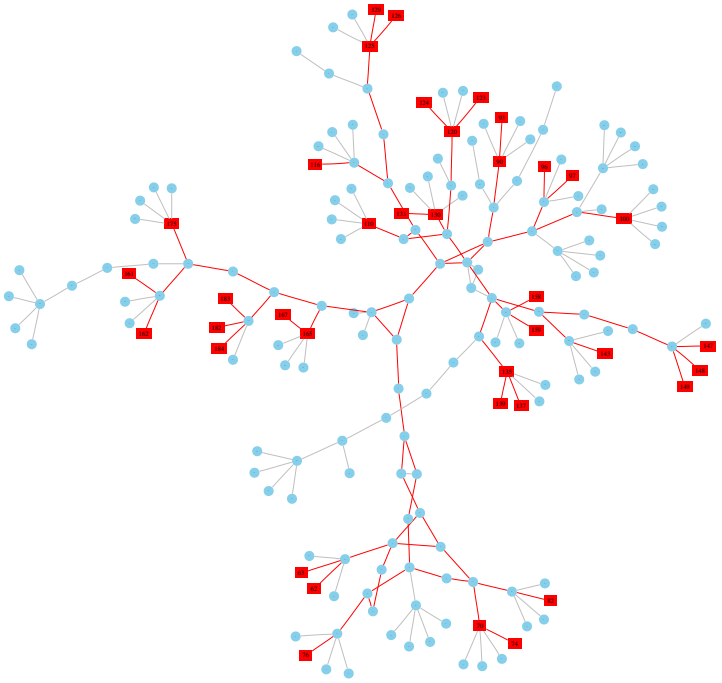


FIG. 3.7 – Première plate-forme. Les rectangles sont les machines sélectionnées. La plate-forme est composée de 38 nœuds, connectés par 49 routeurs et 92 liens de communication.

Les plates-formes sont générées en sélectionnant aléatoirement des nœuds LAN, ensuite un plus court chemin en nombre de sauts est effectué, puis les nœuds induits sont gardés. Ces nœuds sont représentés par des rectangles dans les figures 3.7 et 3.8. Ces rectangles sont les nœuds qui participent au calcul. Ils représentent 30% des ressources initiales. Les autres nœuds sont de simples routeurs. La puissance de calcul des machines sélectionnées est choisie aléatoirement dans une liste de valeurs correspondant à des puissances de calcul (exprimées en Mflops et évaluées par un jeu de tests provenant de LINPACK [11]) d'une grande variété de machines (Pentium Pro 200MHz, Pentium 2 350MHz, Céléron 400MHz, Athlon 1,4GHz, Pentium 4 1,7GHz, ...). Les capacités des arêtes sont attribuées en utilisant la classification du générateur Tiers (liens LAN, LAN/MAN, MAN/WAN, ...). Pour chaque type de lien, nous utilisons des valeurs mesurées à l'aide de `pathchar` [27] entre des machines situées à l'ENS Lyon et d'autres dispersées partout en France (Strasbourg, Lille, Grenoble et

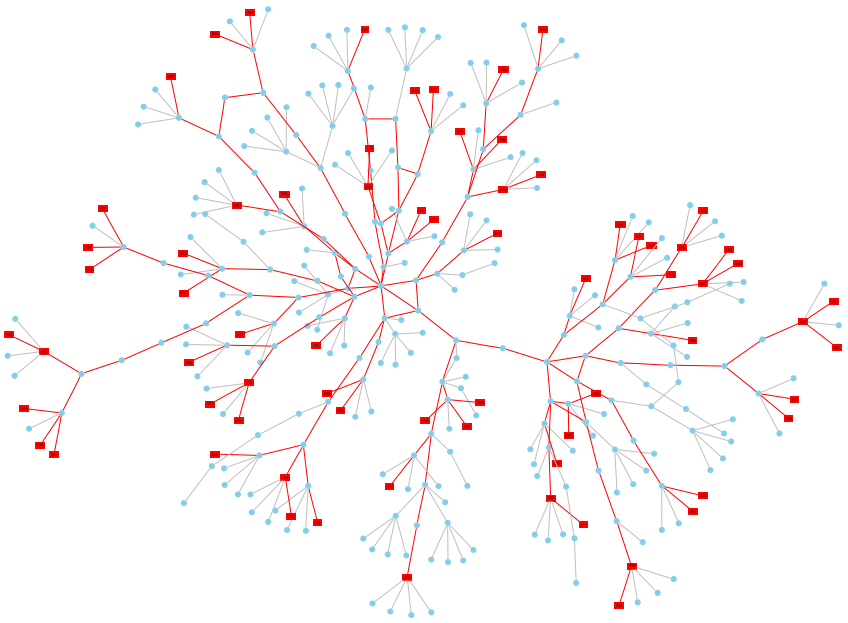


FIG. 3.8 – Deuxième plate-forme. Les rectangles sont les machines sélectionnées. La plate-forme est composée de 90 nœuds, connectés par 43 routeurs et 136 liens de communication.

Orsay), aux États-Unis (Knoxville, San Diego, et Argonne), et au Japon (Nagoya et Tokyo). La seconde plate-forme est deux fois plus grande que la première.

3.4.5.3 Résultats

Nous avons fait varier le ratio D_c/D_w afin de voir quel était le nombre de processeurs impliqués dans la solution optimale. Les figures 3.9 et 3.10 mettent en évidence ce résultat. Les valeurs du ratio D_c/D_w ont été estimées de la manière suivante : supposons que nous avons une matrice carrée de taille n à laquelle nous voulons appliquer quelques filtres. D_c est alors égal à $n \times 64/1024^2$ Mflops et D_w à $n^2/1024^2$ Mflops (s'il y a une opération flottante par cellule) D_c/D_w est alors égal à $64/n$ et pour des valeurs raisonnables de n, le ratio D_c/D_w est compris entre $0,0064$ (pour $n = 10000$) et $0,64$ (pour $n = 100$).

De manière prévisible, la taille de l'anneau augmente lorsque le ratio D_c/D_w décroît. En d'autres termes, plus la quantité de travail est importante vis-à-vis du coût des communications, plus il est intéressant de distribuer le travail. Toutefois, dans les figures 3.11 à 3.16, nous représentons le temps d'exécution normalisé comme une fonction de la taille de l'anneau solution. Pour les deux plates-formes, nous utilisons différents ratio communication-calculs. Pour chaque ratio, il existe une taille optimale d'anneau. Nous pouvons remarquer sur cette série de figures que la taille optimale de l'anneau solution croît plus le ratio diminue. Lorsque D_c/D_w est petit, la forme en « baignoire » des courbes s'explique de la manière suivante : le temps d'exécu-

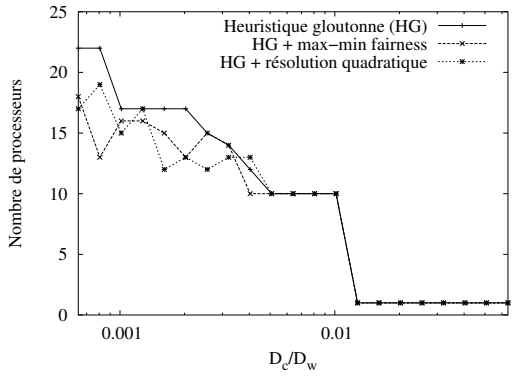


FIG. 3.9 – Première plate-forme. Taille de l'anneau optimal en fonction du ratio D_c/D_w.

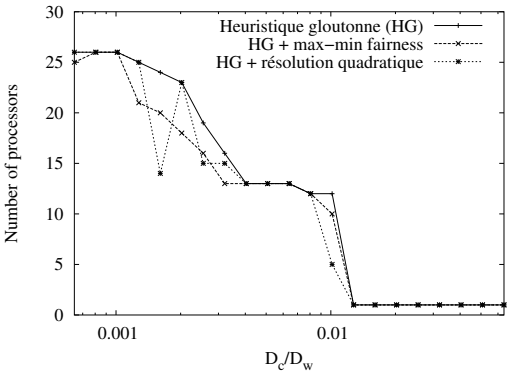


FIG. 3.10 – Deuxième plate-forme. Taille de l'anneau optimal en fonction du ratio D_c/D_w.

tion commence par décroître lorsque le nombre de processeurs augmente, cela étant dû à la charge de travail qui est répartie sur plus de ressources ; le coût relatif des communications devient alors plus important, et ce n'est pas rentable d'utiliser plus de ressources. Notons que lorsque le ratio D_c/D_w est très grand, il n'est pas bon de distribuer le travail sur plusieurs processeurs.

Finalement, nous évaluons l'utilité de deux variantes (le « max-min fairness » et la programmation quadratique), introduites afin d'améliorer l'heuristique. Étonnamment, l'impact des deux variantes n'est pas très significatif (cf. figures 3.17 et 3.18). En fait, elles s'avèrent être même moins efficaces dans la plupart des cas que la version complète de l'heuristique, qui s'avère être peu coûteuse et très efficace.

3.4.6 Conclusion

Le résultat majeur de cette section est la NP-complétude du problème d'optimisation SHAREDRING. Les simulations effectuées sur les différentes plates-formes mettent en évidence les résultats auxquels nous pouvions nous attendre, à savoir la variation du nombre de processeurs dans la solution optimale en fonction du ratio D_c/D_w. Il est alors maintenant intéressant d'évaluer la pertinence de notre heuristique.

3.5 Impact du partage des liens

Nous allons, dans cette section, évaluer l'impact que peut avoir le partage des liens, c'est-à-dire montrer par des simulations que le partage des liens de communication a un impact important sur les stratégies d'équilibrage de charge.

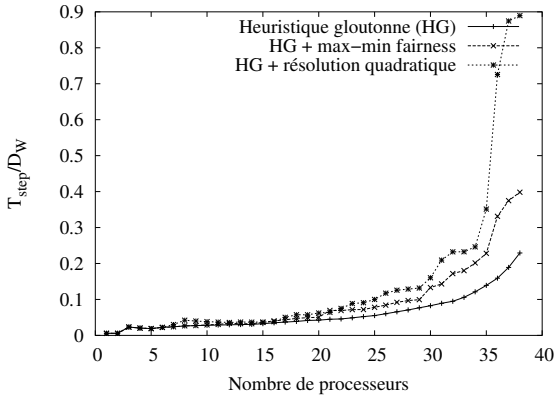


FIG. 3.11 – Première plate-forme.

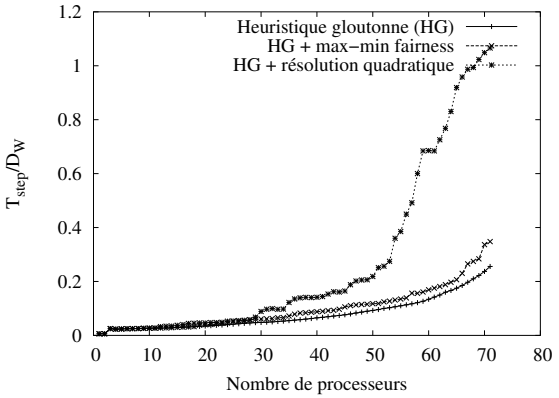


FIG. 3.12 – Deuxième plate-forme.

Figures 3.11 et 3.12 : valeur de T_{step} normalisée par D_w comme une fonction de la taille de l'anneau solution, avec un ratio communication/calcul : $D_c/D_w = 0,064$. Dans les deux cas, l'anneau optimal est de taille 1 : il n'y a pas assez de travail à distribuer, par conséquent ce n'est pas le bon ratio pour permettre de distribuer les données aux différents processeurs.

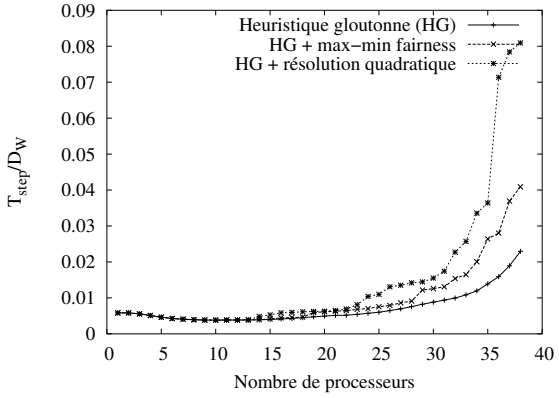


FIG. 3.13 – Première plate-forme. L'anneau le plus efficace trouvé est de taille 10.

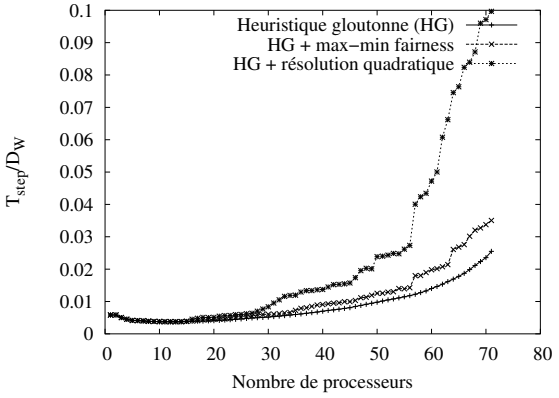


FIG. 3.14 – Deuxième plate-forme. L'anneau le plus efficace trouvé est de taille 13.

Figures 3.13 et 3.14 : valeur de T_{step} normalisée par D_w comme une fonction de la taille de l'anneau solution, avec un ratio communication/calcul : $D_c/D_w = 0,0064$.

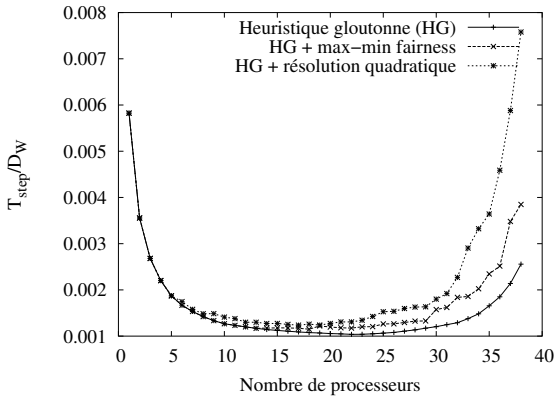


FIG. 3.15 – Première plate-forme. L'anneau le plus efficace trouvé est de taille 22.

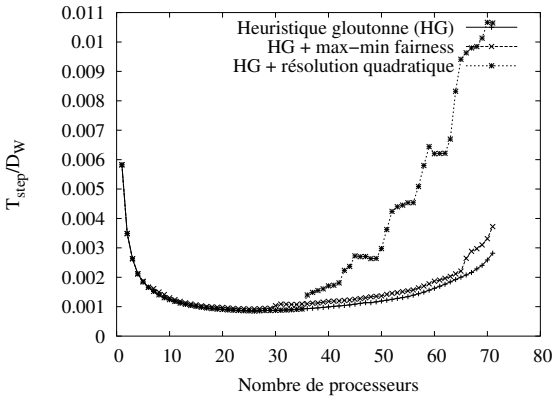


FIG. 3.16 – Deuxième plate-forme. L'anneau le plus efficace trouvé est de taille 26.

Figures 3.15 et 3.16 : valeur de T_{step} normalisée par D_w comme une fonction de la taille de l'anneau solution, avec un ratio communication/calcul : $D_c/D_w = 0,00064$.

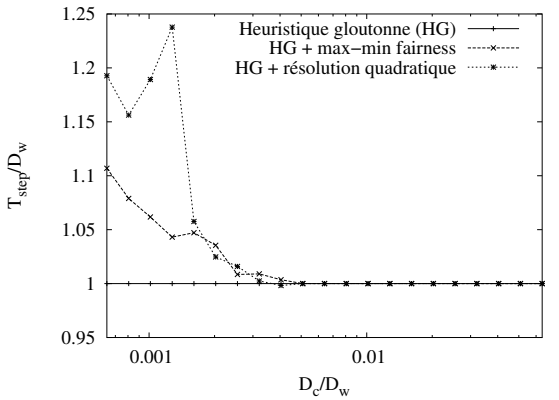


FIG. 3.17 – Première plate-forme.

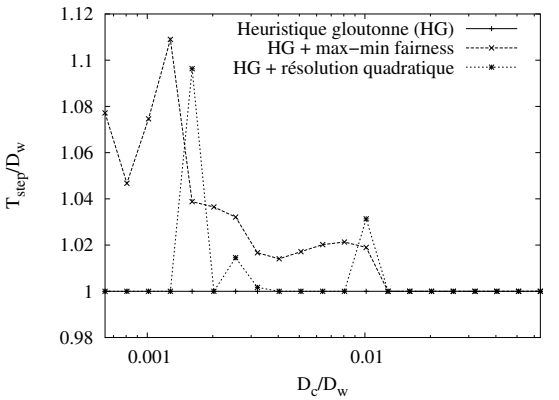


FIG. 3.18 – Deuxième plate-forme.

Figures 3.17 et 3.18 : impact des optimisations sur la qualité de la solution.

3.5.1 Description des plates-formes

Nous avons simulé nos heuristiques sur deux plates-formes, dont les caractéristiques correspondent respectivement aux plates-formes de l'ENS de Lyon et de l'université de Strasbourg. Nous avons aussi utilisé ces deux plates-formes pour nos simulations dans le problème d'optimisation SLICERING (section 3.3.6). Rappelons brièvement leurs caractéristiques. La figure 3.2 montre la plate-forme de Lyon, composée de 14 unités de calcul et 3 routeurs. Une vue abstraite de la plate-forme de Lyon est représentée sur la figure 3.19. Sur cette figure les nœuds 0 à 13, en forme de cercle, sont les processeurs, alors que les nœuds 14 à 16, en forme de losange, sont les routeurs. Les arêtes sont étiquetées avec les bandes passantes des liens. Les temps de cycle des processeurs (en secondes par mégaflop) sont rassemblés dans le tableau 3.1. De façon similaire, la plate-forme de Strasbourg (figure 3.3) est composée de 13 unités de calcul et de 6 routeurs. Une vue abstraite de la plate-forme de Strasbourg est représentée sur la figure 3.20. Les temps de cycle des processeurs sont rassemblés dans le tableau 3.2.

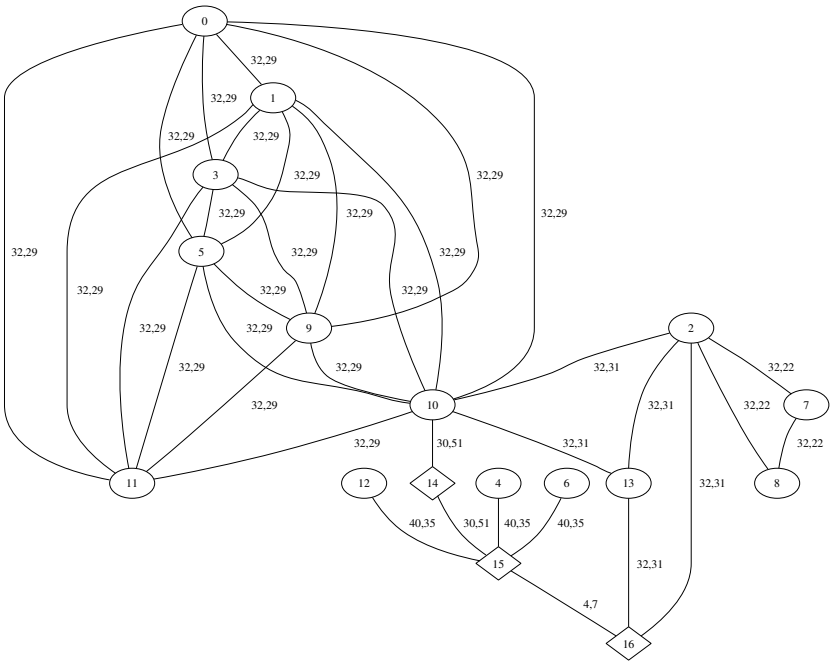


FIG. 3.19 – Vue abstraite de la plate-forme de Lyon.

3.5.2 Simulations

Pour les deux topologies, nous évaluons l'impact du partage de liens de la manière suivante. La première heuristique construit un anneau solution sans prendre en compte le partage de liens. En utilisant le graphe abstrait de la figure 3.19, nous construisons un graphe totalement interconnecté à l'aide d'un algorithme de plus court chemin qui détermine la bande passante entre chaque paire de processeurs. Puis

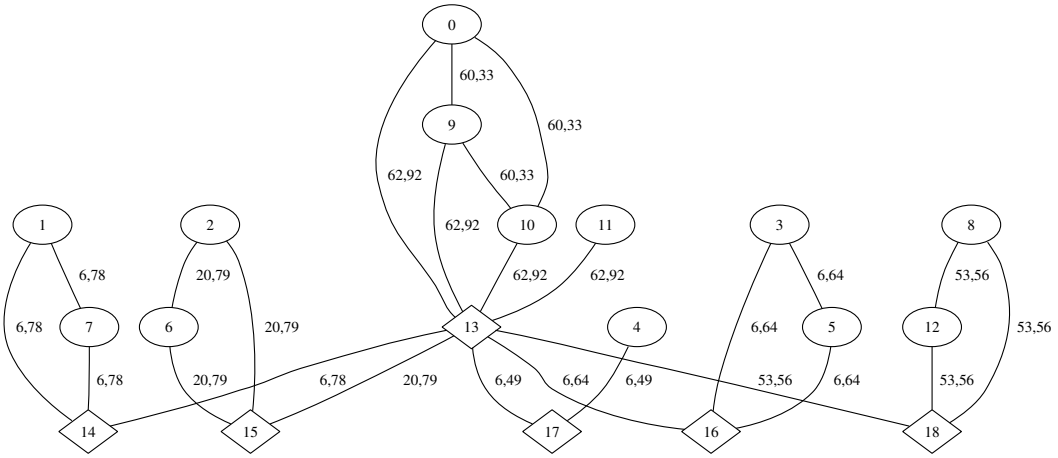


FIG. 3.20 – Vue abstraite de la plate-forme de Strasbourg.

nous retournons l'anneau solution calculé par l'heuristique gloutonne pour le problème SLICERING décrite en Section 3.3.5. La valeur de T_{step} obtenue par l'heuristique peut très bien ne pas être réalisable du fait que le réseau réel n'est pas un graphe complet. De ce fait, nous conservons :

- l'anneau,
- les chemins de communication entre les processeurs adjacents de cet anneau.

Nous évaluons ensuite cette solution en utilisant l'optimisation de programmation quadratique.

La seconde heuristique est l'heuristique gloutonne conçue en Section 3.4.4 pour le problème SHAREDRING, en utilisant l'optimisation de programmation quadratique. La différence majeure entre les deux heuristiques est que la seconde prend en compte la contention des liens lors de la construction de l'anneau.

Pour comparer la valeur de $\frac{T_{step}}{D_w}$ retournée par les deux algorithmes nous utilisons différents rapports des coûts de communication et de calcul (D_c/D_w). Les tableaux 3.5 et 3.6 montrent ces valeurs pour chaque plate-forme. Les conclusions pouvant être tirées de ces expérimentations sont les suivantes :

- Lorsque l'impact du coût des communications est faible, le but principal est d'équilibrer les calculs et les deux heuristiques sont équivalentes.
- Lorsque le rapport D_c/D_w devient plus important, l'effet de la contention des liens devient évident et la solution retournée par la seconde heuristique est bien meilleure.

Pour conclure, nous mettons en avant le fait qu'une modélisation précise des communications a un impact important sur la performance des stratégies d'équilibrage de charge.

Ratio D_c/D_w	H1 : slice-ring	H2 : shared-ring	Amélioration
0,1	3,17	3,15	0,63%
1	3,46	3,22	6,94%
10	10,39	3,9	62,46%

TAB. 3.5 – T_{step}/D_w pour chaque heuristique sur la plate-forme de Lyon.

Ratio D_c/D_w	H1 : slice-ring	H2 : shared-ring	Amélioration
0,1	7,32	7,26	0,82%
1	9,65	7,53	21,97%
10	19,24	10,26	46,67%

TAB. 3.6 – T_{step}/D_w pour chaque heuristique sur la plate-forme de Strasbourg.

3.6 Conclusion

Nous nous sommes intéressée dans ce chapitre aux algorithmes itératifs, d'une part sur une plate-forme homogène complète, et d'autre part sur une plate-forme hétérogène complète puis quelconque. Notre but a été de modéliser de façon réaliste les communications concourantes sur une station de travail et de mettre en évidence l'importance de la contention des liens de communication. Un des résultats est tout d'abord la NP-complétude des problèmes d'optimisation SLICERING et SHAREDRING. Cependant, ce résultat peut apparaître de manière négative. Or, une contribution importante de ce chapitre est la conception d'une heuristique efficace, qui prend pleinement en compte les contentions lors des diverses communications. Cela est clairement mis en évidence par les simulations, d'où l'importance d'une modélisation précise des communications.

Du fait que nous supposons une certaine connaissance sur notre plate-forme hétérogène pour prendre les décisions d'équilibrage de charge, cela engendre une limitation dans notre travail. En fait, les techniques d'équilibrage de charge peuvent être introduites, soit dynamiquement, soit statiquement, soit par un mélange des deux. Il est clair que sur les grandes plates-formes hétérogènes il sera préférable d'appliquer des schémas dynamiques si les variations de performance s'effectuent très rapidement. Cependant, dans un tel contexte, il est possible d'injecter quelques connaissances statiques et ainsi permettre l'obtention d'heuristiques efficaces lors des variations de performance de la plate-forme hétérogène. Nous allons nous intéresser à ce problème dans le chapitre suivant.

Chapitre 4

Redistribution de données

Je vais m'occuper de la redistribution du revenu national à tous ! La foule reprend en choeur : Oui, redistribution, redistribution... ! ...

4.1 Introduction

Nous nous intéressons au problème de redistribution de données sur des anneaux de processeurs homogènes et hétérogènes. Ce problème surgit lorsqu'une phase d'équilibrage de charge doit avoir lieu. Du fait des variations de performances des ressources (puissance de calcul disponible des processeurs, capacité de la bande passante) et des demandes du système (tâches terminées, nouvelles tâches, etc.), les données doivent être redistribuées sur l'ensemble des processeurs participants afin de mieux équilibrer la charge. Nous ne parlerons pas ici du mécanisme d'équilibrage de charge lui-même (nous le considérons comme extérieur, pouvant être un système, un algorithme, un oracle...) puisque cela a été traité dans le chapitre précédent (chapitre 3). Nous visons seulement à optimiser la redistribution de données induite par le mécanisme d'équilibrage de charge.

Nous considérons un ensemble de n processeurs P_1, P_2, ..., P_n. Chaque processeur P_k possède initialement un nombre de données L_k. Le système (l'algorithme ou l'oracle) d'équilibrage de charge décide que la nouvelle charge de P_k doit être $L_k - \delta_k$. Si $\delta_k > 0$, P_k est surchargé et il devra envoyer δ_k données aux autres processeurs ; par contre, si $\delta_k < 0$, P_k n'est pas suffisamment chargé et il devra recevoir $-\delta_k$ données de la part des autres processeurs. Nous avons bien sûr la loi de conservation : $\sum_{k=1}^{n} \delta_k = 0$, qui se traduit par le fait que la charge de travail reste la même sur l'ensemble de l'anneau malgré les variations de performances. Notre but est de déterminer les communications nécessaires et de les réaliser (ce que nous appelons la redistribution de données) en un temps minimal.

Nous supposons que les processeurs participant sont organisés en anneau. Celui-ci peut être soit unidirectionnel, soit bidirectionnel, ce qui signifie qu'un processeur pourra communiquer soit avec un seul de ses voisins, soit avec ses deux voisins dans l'anneau. Par ailleurs les liens de communication seront soit homogènes soit hétérogènes. Ceci impliquera donc quatre cadres de travail différents. Il existe deux principaux contextes dans lesquels les anneaux de processeurs sont utiles : en ce qui concerne le premier, l'ordre des processeurs doit être préservé. Considérons une matrice dont les colonnes sont distribuées aux processeurs avec la condition que chaque processeur travaille sur une tranche consécutive de colonnes. Un processeur surchargé P_i envoie ses premières colonnes au processeur P_j ayant la tranche de colonnes précédant ses propres colonnes (et P_j ajoutera ces colonnes à l'extrémité de sa tranche). De la même manière, P_i envoie ses dernières colonnes au processeur ayant la tranche suivante de colonnes. Ce sont les seules possibilités. En d'autres termes, la distribution de données linéaire appelle à un arrangement des processeurs unidimensionnel, c'est-à-dire à un arrangement en anneau.

Le second contexte qui peut amener à utiliser un anneau est la simplicité au niveau de la programmation. En utilisant un anneau, unidirectionnel ou bidirectionnel, la gestion des données à redistribuer est plus simple. Des intervalles de données peuvent être maintenus et mis-à-jour pour caractériser la charge de chaque processeur. Finalement, nous observons que les machines parallèles avec une topologie riche mais fixe d'interconnexion (hypercubes, arbres, grilles, pour en citer quelques-uns) sont sur le déclin. Les architectures de grappes hétérogènes, que nous utilisons dans cet article, ont un large graphe d'interconnection inconnu possédant des *backbones*, des passerelles et des commutateurs, et modéliser le graphe de communication comme un anneau est un choix raisonnable.

Comme il a été dit précédemment, nous allons étudier quatre cas d'algorithmes de redistribution. Dans le cas le plus simple, c'est-à-dire un anneau homogène unidirectionnel, nous avons obtenu un algorithme optimal. Comme notre architecture est simple, nous sommes en mesure de fournir des formules (analytiques) explicites du nombre de données envoyées et reçues par chaque processeur. Le même résultat est obtenu dans le cas d'un anneau homogène bidirectionnel mais l'algorithme devient plus compliqué. En supposant les liens de communications hétérogènes, nous obtenons un algorithme optimal dans le cas unidirectionnel mais avec une formulation asynchrone. Cependant, nous devons recourir à une heuristique basée sur la programmation linéaire pour le cas bidirectionnel hétérogène.

4.2 Modèle

Nous considérons un ensemble de n processeurs $P_1, P_2, \ldots, P_n$ organisés en anneau. Le successeur de P_i dans l'anneau est P_{i+1}, et son prédécesseur est P_{i-1}, tous les indices étant pris modulo n. Pour $1 \leqslant k, l \leqslant n$, $C_{k,l}$ est la *tranche* de processeurs consécutifs $C_{k,l} = P_k, P_{k+1}, \ldots, P_{l-1}, P_l$.

Comme auparavant, nous notons $c_{i,i+1}$ la capacité du lien (logique) de communication de P_i à P_{i+1}. En d'autres termes, nous avons besoin de $c_{i,i+1}$ unités de temps pour envoyer une donnée du processeur P_i au processeur P_{i+1}. Dans le cas d'un anneau bidirectionnel, $c_{i,i-1}$ est la capacité du lien de P_i à P_{i-1}. Nous utilisons le modèle 1-port pour les communications : à un instant donné, il y a au plus deux communications impliquant un processeur donné, un unique envoi et une unique réception. Ce modèle permet une vue réaliste des plates-formes de calcul puisque les nœuds de ces plates-formes ne sont pas capables d'envoyer (ou de recevoir) des données simultanément. Un processeur peut à la fois envoyer et recevoir une donnée, ce qui n'implique aucune restriction dans le cas unidirectionnel ; cependant, dans le cas bidirectionnel, un processeur ne peut pas envoyer simultanément une donnée à son successeur et à son prédécesseur ; ni recevoir simultanément une donnée de chaque voisin. Ceci est la seule restriction induite par le modèle : n'importe quelle paire de communications qui ne viole pas la contrainte du modèle 1-port peut avoir lieu en parallèle.

Chaque processeur P_k possède initialement L_k données. Après redistribution, P_k possédera $L_k - \delta_k$ données. Nous appelons δ_k le *déséquilibre* de P_k. Nous notons $\delta_{k,l}$ le déséquilibre total de la tranche de processeurs $C_{k,l}$: $\delta_{k,l} = \delta_k + \delta_{k+1} + \cdots + \delta_{l-1} + \delta_l$.

Du fait de la loi de conservation des données, $\sum_{k=1}^{n} \delta_k = 0$. Évidemment le déséquilibre d'un processeur ne peut pas être plus grand que sa charge initiale : $L_k \geqslant \delta_k$. En fait, nous supposons que chaque processeur possède au moins une donnée initialement ($L_k \geqslant 1$), et après redistribution ($L_k \geqslant 1 + \delta_k$) : sinon, nous devrions construire le nouvel anneau du sous-ensemble des processeurs toujours impliqués dans le calcul.

4.3 Anneau homogène unidirectionnel

Nous considérons ici le cas de l'anneau homogène unidirectionnel. Tout processeur P_i peut seulement envoyer des données à son successeur P_{i+1}, et $c_{i,i+1} = c$ pour tout $i \in [1, n]$. Nous établissons d'abord une borne inférieure sur le temps d'exécution des algorithmes de redistribution de données, puis nous présentons un algorithme (optimal) atteignant cette borne, et nous prouvons son exactitude.

4.3.1 Borne inférieure du temps d'exécution

Lemme 4.1. *Soit τ le temps optimal de la redistribution. Alors :*

$$\tau \geqslant \left(\max_{1 \leqslant k \leqslant n,\ 0 \leqslant l \leqslant n-1} |\delta_{k,k+l}| \right) \times c. \tag{4.1}$$

Démonstration. La tranche de processeurs $C_{k,k+l} = P_k, P_{k+1}, \ldots, P_{k+l-1}, P_{k+l}$ a un déséquilibre total de $\delta_{k,k+l} = \delta_k + \delta_{k+1} + \cdots + \delta_{k+l-1} + \delta_{k+l}$. Si $\delta_{k,k+l} > 0$, $\delta_{k,k+l}$ données doivent être envoyées de $C_{k,k+l}$ vers les autres processeurs. L'anneau est unidirectionnel, donc P_{k+l} est le seul processeur dans $C_{k,k+l}$ avec un lien sortant. Donc, P_{k+l} a besoin d'un

temps égal à $\delta_{k,k+l} \times c$ pour envoyer $\delta_{k,k+l}$ données. Par conséquent, une redistribution totale des données ne peut pas prendre moins de $\delta_{k,k+l} \times c$. Le raisonnement est similaire pour le cas $\delta_{k,k+l} < 0$. ■

4.3.2 Algorithme optimal

Nous présentons l'algorithme suivant de redistribution qui commence par repérer une tranche de processeurs définissant la borne inférieure du lemme 4.1 et dont l'ensemble des envois est déterminé à partir de cette borne inférieure :

Algorithme 4.1 Algorithme de redistribution pour anneaux homogènes unidirectionnels

1: Soit $\delta_{\max} = (\max_{1 \leqslant k \leqslant n, 0 \leqslant l \leqslant n-1} |\delta_{k,k+l}|)$
2: Soient `start` et `end` deux indices tels que la tranche $C_{\texttt{start,end}}$ est de déséquilibre maximal : $\delta_{\texttt{start,end}} = \delta_{\max}$.
3: **Pour** $s = 1$ à $\delta_{\max}$ **:**
4: **Pour tout** $l = 0$ à $n - 1$ **:**
5: **Si** $\delta_{\texttt{start,start}+l} \geqslant s$ **Alors**
6: $P_{\texttt{start}+l}$ envoie à $P_{\texttt{start}+l+1}$ une donnée pendant l'intervalle de temps $[(s - 1) \times c, s \times c[$

Nous commençons par prouver l'exactitude de l'algorithme 4.1 (lemme 4.3), puis nous prouvons son optimalité (lemme 4.4). Intuitivement, si l'étape 6 est toujours faisable, alors chaque exécution de l'étape 3 prend un temps c et l'algorithme atteint le temps d'exécution du lemme 4.1.

Tout d'abord, nous vérifions que la tranche $C_{\texttt{start,end}}$ est bien définie à l'étape 2 de l'algorithme : pour toute tranche ayant un déséquilibre δ, la tranche composée des processeurs restants a le déséquilibre opposé $-\delta$. Ensuite, nous énonçons le rôle particulier du processeur $P_{\texttt{start}}$:

Lemme 4.2. *Le processeur $P_{\texttt{start}}$ ne reçoit aucune donnée pendant le temps d'exécution de l'algorithme 4.1.*

Démonstration. Nous prouvons ce résultat par l'absurde. Supposons que pour une itération s donnée, le processeur $P_{\texttt{start}}$ reçoive une donnée. Alors, le prédécesseur de $P_{\texttt{start}}$ dans l'anneau, $P_{\texttt{start}-1}$, envoie au moins une donnée lors de cette itération. $P_{\texttt{start}-1}$ devient alors émetteur avec la condition de l'étape 5 de l'algorithme 4.1, $\sum_{j=0}^{n-1} \delta_{\texttt{start}+j} = \delta_{\texttt{start,start}-1} \geqslant s$. Cependant, avec la loi de conservation, $\sum_{i=1}^{n} \delta_i = 0$ et donc $0 \geqslant s$, ce qui nous permet d'obtenir la contradiction recherchée. ■

Pour prouver l'exactitude de l'algorithme 4.1, nous devons montrer que pendant chaque itération, n'importe quel processeur possède au moins une donnée à envoyer à l'étape 6. En d'autres termes, nous devons prouver qu'aucun processeur n'envoie une

donnée qu'il n'a pas en sa possession. Soit L_i^s la charge de P_i à la fin de l'itération s de l'algorithme 4.1 :

Lemme 4.3. *Pendant l'itération s de la boucle 3, si P_i envoie une donnée, alors $L_i^{s-1} \geqslant 1$.*

Démonstration. Nous prouvons le lemme 4.3 par récurrence. Par définition du déséquilibre (cf. section 4.2), nous savons que chaque processeur P_i possède initialement $L_i^0 = L_i \geqslant 1$ données. Le résultat est vrai au rang 1 ($s = 1$). Supposons le résultat vrai jusqu'à l'itération s inclue et démontrons qu'il est vrai à l'itération $s+1$. Il faut considérer deux cas, suivant si le processeur P_i est supposé recevoir ou non une donnée pendant l'itération $s+1$:

1. Si P_i est à la fois émetteur et récepteur pendant l'itération $s+1$, alors P_i est à la fois émetteur et récepteur pendant l'itération s à cause de la condition de l'étape 5 de l'algorithme 4.1. La charge de P_i après l'itération s est alors la même que précédemment et $L_i^s = L_i^{s-1}$. Nous concluons en utilisant l'hypothèse de récurrence.
2. Si P_i est émetteur mais pas récepteur pendant l'itération $s+1$, nous devons vérifier que P_i n'envoie pas une donnée qu'il ne possède pas. Puisque P_i est émetteur, nous avons par la condition de l'étape 5 de l'algorithme 4.1 :

$$\delta_{\texttt{start},i} \geqslant s+1. \tag{4.2}$$

Donc, P_i a envoyé une donnée à chaque itération précédente.

Pendant l'itération $s+1$, P_i n'est pas récepteur. Donc P_{i-1} n'est pas émetteur pendant cette itération, et, par la condition de l'étape 5 de l'algorithme 4.1, nous avons : $\delta_{\texttt{start},i-1} < s+1$. Pendant chaque itération de 1 à $\delta_{\texttt{start},i-1}$, P_{i-1} a envoyé une donnée (cf. ci-dessous pour la preuve que $\delta_{\texttt{start},\texttt{start}+j} \geqslant 0$ pour tout $j \in [0, n-1]$). Par conséquent, pendant chacune de ces itérations, P_i était à la fois émetteur et récepteur, et ni sa charge ni son déséquilibre n'ont changé. Pendant chaque itération de $1 + \delta_{\texttt{start},i-1}$ à s, le processeur P_i était émetteur mais pas récepteur. Ainsi sa charge et son déséquilibre ont diminué d'un pendant chacune de ces itérations. Par conséquent :

$$L_i^s = L_i - (s - \delta_{\texttt{start},i-1}). \tag{4.3}$$

Cependant, $\delta_i + \delta_{\texttt{start},i-1} = \delta_{\texttt{start},i}$. Donc, l'équation 4.3 est équivalente à : $L_i^s = L_i - \delta_i + \delta_{\texttt{start},i} - s$. Par l'équation 4.2, nous obtenons que $\delta_{\texttt{start},i} - s \geqslant 1$. Or, par hypothèse (section 4.2), nous avons supposé que $L_i \geqslant 1 + \delta_i$. D'où $L_i^s \geqslant 2$.

La preuve ci-dessus se base sur la propriété que pour toute valeur de $j \in [0, n-1]$, $\delta_{\texttt{start},\texttt{start}+j} \geqslant 0$. Nous allons prouver ce résultat par l'absurde. Supposons qu'il existe une valeur j telle que $\delta_{\texttt{start},\texttt{start}+j} < 0$. Il faut considérer deux cas :

1. $j + \texttt{start} \in [\texttt{start}, \texttt{end}]$. Alors $\delta_{\texttt{start},\texttt{end}} = \delta_{\texttt{start},\texttt{start}+j} + \delta_{\texttt{start}+j+1,\texttt{end}}$ et $\delta_{\texttt{start},\texttt{end}} < \delta_{\texttt{start}+j+1,\texttt{end}}$ ce qui contredit la maximalité de $C_{\texttt{start},\texttt{end}}$.

2. $j + \texttt{start} \notin [\texttt{start}, \texttt{end}]$. Alors $\delta_{\texttt{start},j+\texttt{start}} = \delta_{\texttt{start},\texttt{end}} + \delta_{1+\texttt{end},j+\texttt{start}}$. Donc $\delta_{\texttt{start},\texttt{end}} < -\delta_{1+\texttt{end},j+\texttt{start}}$. Or, la somme totale des déséquilibres étant nulle par définition, la somme des déséquilibres de $C_{1+\texttt{end},j+\texttt{start}}$ est égale à l'opposé de la somme des déséquilibres de $C_{j+1+\texttt{start},\texttt{end}}$. Par conséquent, $\delta_{\texttt{start},\texttt{end}} < \delta_{j+1+\texttt{start},\texttt{end}}$, ce qui contredit la maximalité de $C_{\texttt{start},\texttt{end}}$.

■

Nous avons prouvé l'exactitude de l'algorithme 4.1. Il nous reste à prouver que lorsqu'il se termine, la redistribution entière a été effectuée :

Lemme 4.4. *Lorsque l'algorithme 4.1 termine après l'itération δ_{max}, c'est-à-dire au temps τ, la charge de tout processeur P_i est égale à $L_i - \delta_i$.*

Démonstration. Nous prouvons par récurrence sur les indices de processeur, en commençant au processeur $P_{\texttt{start}}$, que tout processeur P_j a une charge $L_j - \delta_j$ pour toute itération $s \geqslant \max_{0 \leqslant i \leqslant j} \delta_{\texttt{start},\texttt{start}+i}$.

Comme il a été établi par le lemme 4.2, le processeur $P_{\texttt{start}}$ ne reçoit jamais de données au cours de l'exécution. Donc, après $\delta_{\texttt{start},\texttt{start}} = \delta_{\texttt{start}}$ itérations de la boucle 3, $P_{\texttt{start}}$ n'est jamais le récepteur ni l'émetteur d'une donnée. $P_{\texttt{start}}$ possède exactement $L_{\texttt{start}} - \delta_{\texttt{start}}$ données, c'est-à-dire sa charge initiale moins les données qu'il avait à envoyer.

Nous supposons le résultat vrai jusqu'au processeur $P_{\texttt{start}+l}$ (avec $l \geqslant 0$) inclus. Démontrons le résultat pour le processeur $P_{\texttt{start}+l+1}$. En utilisant l'hypothèse de récurrence, nous savons que pour toute itération $s \geqslant \max_{0 \leqslant i \leqslant l} \delta_{\texttt{start},\texttt{start}+i}$, la charge totale de la tranche $C_{\texttt{start},\texttt{start}+l}$ est égale à $\sum_{0 \leqslant i \leqslant l} L_i - \sum_{0 \leqslant i \leqslant l} \delta_i$.

Pendant l'exécution de l'algorithme, le processeur $P_{\texttt{start}+l+1}$ a envoyé exactement $\delta_{\texttt{start},\texttt{start}+l+1}$ données. Toutes ces opérations d'envoi ont lieu avant ou pendant l'itération $\delta_{\texttt{start},\texttt{start}+l+1}$. De plus, le lemme 4.2 établit que le processeur $P_{\texttt{start}}$ ne reçoit jamais de données pendant l'exécution. Donc, la charge totale de la tranche $C_{\texttt{start},\texttt{start}+l+1}$ ne change plus après l'itération $\delta_{\texttt{start},\texttt{start}+l+1}$, et sa charge totale est égale à sa charge initiale moins les données envoyées par le processeur $P_{\texttt{start}+l+1}$: $(\sum_{0 \leqslant i \leqslant l+1} L_i) - \delta_{\texttt{start},\texttt{start}+l+1}$. Donc, après toute itération s, où $s \geqslant \max\{\max_{0 \leqslant i \leqslant l} \delta_{\texttt{start},\texttt{start}+i}, \delta_{\texttt{start},\texttt{start}+l+1}\} = \max_{0 \leqslant i \leqslant l+1} \delta_{\texttt{start},\texttt{start}+i}$, nous connaissons la charge totale des deux tranches de processeurs $C_{\texttt{start},\texttt{start}+l}$ et $C_{\texttt{start},\texttt{start}+l+1}$. Et donc, nous connaissons la charge de $P_{\texttt{start}+l+1}$:

$$\begin{aligned} L^t_{\texttt{start}+l+1} &= \left((\sum_{0 \leqslant i \leqslant l+1} L_{\texttt{start}+i}) - \delta_{\texttt{start},\texttt{start}+l+1} \right) - (\sum_{0 \leqslant i \leqslant l} L_{\texttt{start}+i} - \sum_{0 \leqslant i \leqslant l} \delta_{\texttt{start}+i}) \\ &= L_{\texttt{start}+l+1} - \delta_{\texttt{start}+l+1}. \end{aligned} \tag{4.4}$$

Pour conclure, nous avons juste besoin de remarquer que $\delta_{\max} = \max_{0 \leqslant i \leqslant n-1} \delta_{\texttt{start},\texttt{start}+i}$. ■

L'optimalité de l'algorithme 4.1 est une conséquence directe des lemmes précédents :

Théorème 4.1. *L'algorithme 4.1 est optimal.*

4.4 Anneau hétérogène unidirectionnel

Dans cette section, nous supposons que l'anneau est unidirectionnel mais nous ne supposons plus que les chemins de communication ont la même capacité. Nous nous sommes appuyée sur les résultats de la section précédente afin d'obtenir un algorithme optimal (Algorithme 4.2). Dans cet algorithme, le nombre de données envoyées par un processeur P_i est exactement le même que dans l'algorithme 4.1 ($\delta_{\texttt{start},i}$) . Cependant, comme les liens de communication n'ont plus les mêmes capacités, nous n'avons plus un comportement synchrone. Un processeur P_i envoie ses $\delta_{\texttt{start},i}$ données dès que possible, et de ce fait nous ne pouvons pas exprimer son temps d'exécution avec une formule simple. En effet, si P_i possède initialement plus de données qu'il doit en envoyer, nous aurons le même comportement que précédemment : P_i envoie ses données pendant l'intervalle de temps $[0, \delta_{\texttt{start},i} \times c_{i,i+1}[$. Par contre, si P_i possède moins de données que ce qu'il doit envoyer ($L_i < \delta_{\texttt{start},i}$), P_i commence à envoyer ses données au temps 0 mais devra attendre d'avoir reçu d'autres données de P_{i-1} pour pouvoir les transmettre à P_{i+1}.

Algorithme 4.2 Algorithme de redistribution pour anneaux hétérogènes unidirectionnels

1: Soit $\delta_{\text{max}} = (\max_{1 \leqslant k \leqslant n, 0 \leqslant l \leqslant n-1} |\delta_{k,k+l}|)$
2: Soient `start` et `end` deux indices tels que la tranche $C_{\texttt{start},\texttt{end}}$ est de déséquilibre maximal : $\delta_{\texttt{start},\texttt{end}} = \delta_{\text{max}}$.
3: **Pour tout** $l = 0$ à $n - 1$ **:**
4: $\quad P_{\texttt{start}+l}$ envoie $\delta_{\texttt{start},\texttt{start}+l}$ données une par une et dès que possible au processeur $P_{\texttt{start}+l+1}$

Le fait que l'algorithme 4.2 soit asynchrone implique son exactitude par construction : nous attendons de recevoir une donnée avant de l'envoyer. Donc, lorsque l'algorithme se termine, la redistribution est complète.

Lemme 4.5. *Le temps d'exécution de l'algorithme 4.2 est*

$$\max_{0 \leqslant l \leqslant n-1} \delta_{start,start+l} \times c_{start+l,start+l+1}.$$

Le résultat du lemme 4.5 est surprenant. Intuitivement, il dit que le temps d'exécution de l'algorithme 4.2 est égal au maximum des temps de communication de tous les processeurs, dans le cas où chacun d'eux aurait initialement en local toutes les données qu'il doit envoyer durant l'exécution de l'algorithme. En d'autres termes, il n'y

a aucun délai dû à l'attente, par un processeur, de la réception des données qu'il doit retransmettre.

Démonstration. Nous prouvons ce résultat par contradiction, en supposant que le temps d'exécution de l'algorithme 4.2, noté $t_{\max}$, est plus grand que $\max_{0 \leqslant l \leqslant n-1} \delta_{\texttt{start},\texttt{start}+l} \times c_{\texttt{start}+l,\texttt{start}+l+1}$ (nous supposons que l'algorithme commence à s'exécuter au temps 0). Soit P_i un processeur dont le temps d'exécution est $t_{\max}$, c'est-à-dire que P_i est un processeur qui termine l'émission de sa dernière donnée à la date $t_{\max}$. Par hypothèse, $t_{\max} > \delta_{\texttt{start},i} \times c_{i,i+1}$. Il y a donc des périodes pendant l'exécution de l'algorithme durant lesquelles le processeur P_i n'envoie pas de données au processeur P_{i+1}. Soit t_i la *dernière* date pendant à laquelle P_i n'envoie pas de données. Alors, par définition de t_i, de la date t_i jusqu'à la fin d'exécution de l'algorithme, le processeur P_i envoie en continu de données à P_{i+1}. Soit n_i le nombre de données que P_i envoie pendant l'intervalle de temps. Notons que nous avons $t_{\max} = t_i + n_i \times c_{i,i+1}$. Nous prouvons par récurrence que pour toute valeur de $j \geqslant 1$:

1. Le processeur P_{i-j} envoie une donnée au processeur P_{i-j+1} pendant l'intervalle de temps
   $[t_i - \sum_{k=1}^{j} c_{i-k,i-k+1}, t_i - \sum_{k=1}^{j-1} c_{i-k,i-k+1}]$.
2. Entre la date $t_i - \sum_{k=1}^{j} c_{i-k,i-k+1}$ et la fin de l'exécution de l'algorithme, le processeur P_{i-j} envoie au moins $j + n_i$ données au processeur P_{i-j+1}.
3. $c_{i-j,i-j+1} \leqslant c_{i,i+1}$.
4. Juste avant la date $t_i - \sum_{k=1}^{j} c_{i-k,i-k+1}$, le processeur P_{i-j} n'envoie pas de données au processeur P_{i-j+1}.

Une fois que nous avons prouvé ces propriétés, la contradiction que nous cherchons à obtenir est obtenue grâce au processeur $P_{\texttt{start}}$. Le processeur $P_{\texttt{start}}$ envoie seulement des données qu'il possède initialement ($\delta_{\texttt{start}} = \delta_{\texttt{start},\texttt{start}} \leqslant L_{\texttt{start}}$), et ne reçoit aucune donnée de son prédécesseur dans l'anneau. Cependant, en utilisant les propriétés précédentes, il existe une valeur de $j \geqslant 0$ telle que $\texttt{start} = i - j$, et entre la date $t_i - \sum_{k=1}^{j+1} c_{i-k,i-k+1}$ et la fin d'exécution de l'algorithme, le processeur P_{i-j-1} envoie au moins $j + 1 + n_i$ données au processeur $P_{i-j} = P_{\texttt{start}}$. D'où la contradiction.

La construction utilisée dans la preuve est illustrée par la figure 4.1. Nous commençons par prouver les propriétés souhaitées pour $j = 1$.

1. Par définition de t_i, le processeur P_i n'envoie pas de données au processeur P_{i+1} juste avant la date t_i. À cause de la nature « dès que possible » de l'algorithme, le processeur P_i n'a pas en sa possession une seule donnée juste avant la date t_i et il attend que le processeur P_{i-1} lui en envoie une. Ainsi, la donnée que le processeur P_i commence à envoyer à la date t_i lui est envoyée par le processeur P_{i-1} pendant l'interval de temps $[t_i - c_{i-1,i}, t_i]$.
2. Entre la date t_i et la fin d'exécution de l'algorithme, le processeur P_i envoie n_i données au processeur P_{i+1}. Par hypothèse, le processeur P_i possède au moins une donnée après l'exécution complète de l'algorithme. Comme P_i ne possède

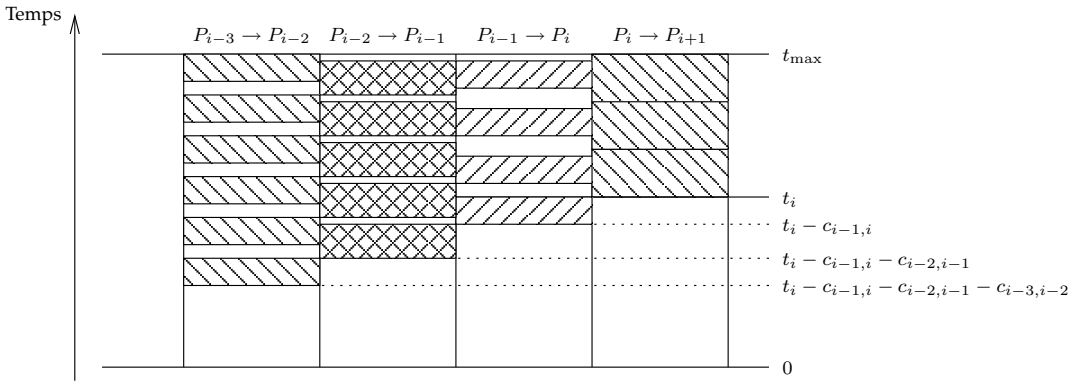


FIG. 4.1 – La construction utilisée dans la preuve du lemme 4.5.

aucune donnée juste avant la date t_i, alors entre les dates $t_i - c_{i-1,i}$ et $t_{\max}$, P_{i-1} envoie au moins $1 + n_i$ données à P_i.

3. De ce qui précède et en utilisant la relation entre t_i, n_i, et $t_{\max}$, nous avons l'implication suivante :

$$t_i + n_i \times c_{i,i+1} = t_{\max} \geqslant (t_i - c_{i-1,i}) + (1 + n_i) \times c_{i-1,i} \qquad \Rightarrow \qquad c_{i,i+1} \geqslant c_{i-1,i}$$

puisque n_i est non nul par définition.

4. Supposons que le processeur P_{i-1} envoie une donnée au processeur P_i juste avant la date $t_i - c_{i-1,i}$. Alors, au plus tôt, cette donnée est reçue par le processeur P_i à la date $t_i - c_{i-1,i}$. Du fait de la nature « dès que possible » de l'algorithme, P_i transmet cette donnée au processeur P_{i+1} (puisqu'il transmet des données reçues plus tard). P_i termine de transmettre cette donnée à la date $t_i - c_{i-1,i} + c_{i,i+1} \geqslant t_i$ au plus tôt. Ainsi, le processeur P_i n'a pas de raison de ne pas envoyer de données à la date t_i, ce qui contredit la définition de t_i.

Nous allons maintenons traiter le cas général par récurrence. Nous supposons que la propriété est vraie jusqu'au processeur P_{i-j} inclus (avec $j \geqslant 1$).

1. Par hypothèse de récurrence, le processeur P_{i-j} n'envoie pas de données au processeur P_{i-j+1} juste avant la date $t_i - \sum_{k=1}^{j} c_{i-k,i-k+1}$. À cause de la nature « dès que possible » de l'algorithme, le processeur P_{i-j} ne possède pas une seule donnée juste avant cette date et il en attend une du processeur P_{i-j-1}. Ainsi, la donnée que le processeur P_{i-j} commence à envoyer à la date $t_i - \sum_{k=1}^{j} c_{i-k,i-k+1}$ lui est envoyé par le processeur P_{i-j-1} pendant l'intervalle de temps $[t_i - \sum_{k=1}^{j+1} c_{i-k,i-k+1}, t_i - \sum_{k=1}^{j} c_{i-j,i-j+1}]$.
2. Entre la date $t_i - \sum_{k=1}^{j} c_{i-k,i-k+1}$ et la fin de l'exécution de l'algorithme, le processeur P_{i-j} envoie $j + n_i$ données au processeur P_{i-j+1}, par hypothèse de récurrence. Par hypothèse, le processeur P_{i-j} possède au moins une donnée après l'exécution de l'algorithme. Comme P_{i-j} ne possède aucune donnée juste avant la date $t_i - \sum_{k=1}^{j} c_{i-k,i-k+1}$, alors entre les dates $t_i - \sum_{k=1}^{j+1} c_{i-k,i-k+1}$ et $t_{\max}$, P_{i-j-1} envoie au moins $1 + j + n_i$ données à P_{i-j}.

3. De ce qui précède et en utilisant la relation entre t_i, n_i, et $t_{\max}$, nous avons l'implication suivante :

$$t_i + n_i \times c_{i,i+1} = t_{\max} \geqslant \left(t_i - \sum_{k=1}^{j+1} c_{i-k,i-k+1} \right) + (1 + j + n_i) \times c_{i-j-1,i-j} \quad \Rightarrow$$

$$n_i \times c_{i,i+1} + \sum_{k=1}^{j} c_{i-k,i-k+1} \geqslant (j + n_i) \times c_{i-j-1,i-j} \quad \Rightarrow$$

$$c_{i,i+1} \geqslant c_{i-j-1,i-j}$$

comme, par hypothèse de récurrence, pour tout $k \in [1, j]$, $c_{i,i+1} \geqslant c_{i-k,i-k+1}$.

4. Supposons que le processeur P_{i-j-1} envoie une donnée au processeur P_{i-j} juste avant la date $t_i - \sum_{k=1}^{j+1} c_{i-k,i-k+1}$. Alors, au plus tôt, cette donnée est reçue par le processeur P_{i-j} à la date $t_i - \sum_{k=1}^{j+1} c_{i-k,i-k+1}$. Du fait de la nature « dès que possible » de l'algorithme, P_{i-j} transmet cette donnée au processeur P_{i-j+1} (puisqu'il transmet des données reçues plus tard). P_{i-j} termine de transmettre cette donnée à la date $t_i - c_{i-j-1,i-j} - \sum_{k=1}^{j-1} c_{i-k,i-k+1}$ au plus tôt. Alors, en suivant le même raisonnement, le processeur P_{i-j+1} la transmet à P_{i-j+2}, qui la reçoit au plus tôt à la date $t_i - c_{i-j-1,i-j} - \sum_{k=1}^{j-2} c_{i-k,i-k+1}$, et ainsi de suite. Donc, le processeur P_i reçoit cette donnée au plus tôt à la date $t_i - c_{i-j-1,i-j}$, et la transmet. Alors, il termine de l'envoyer au plus tôt à la date $t_i - c_{i-j-1,i-j} + c_{i,i+1} \geqslant t_i$, comme nous avons vu que $c_{i,i+1} \geqslant c_{i-j-1,i-j}$. Ainsi, le processeur P_i n'a pas de raison de ne pas envoyer des données à la date t_i, ce qui contredit la définition de t_i. Par conséquent, le processeur P_{i-j-1} n'envoie pas de données au processeur P_{i-j} juste avant la date $t_i - \sum_{k=1}^{j+1} c_{i-k,i-k+1}$.

■

Théorème 4.2. *L'algorithme 4.2 est optimal.*

Démonstration. Soit τ le temps optimal de redistribution. En suivant les arguments utilisés dans la preuve du lemme 4.1 pour le cas homogène, nous obtenons la borne inférieure suivante :

$$\tau \geqslant \max_{1 \leqslant k \leqslant n, 0 \leqslant l \leqslant n-1} |\delta_{k,k+l}| \times c_{k+l,k+l+1}.$$

Nous concluons en utilisant le lemme 4.5. ■

4.5 Anneau homogène bidirectionnel

Nous considérons ici le cas de l'anneau homogène bidirectionnel. Tous les liens ont ici la même capacité mais un processeur peut envoyer des données à ses deux voisins dans l'anneau : il existe une constante c telle que, pour tout $i \in [1, n]$, $c_{i,i+1} = c_{i,i-1} = c$. Nous procédons de la même manière que dans le cas de l'anneau homogène unidirectionnel : nous obtenons une borne inférieure du temps d'exécution de la redistribution de données ainsi qu'un algorithme optimal atteignant cette borne.

4.5.1 Borne inférieure du temps d'exécution

La borne définie par le lemme 4.6 est basée sur la même intuition que celle du lemme 4.1 en ce qui concerne la première partie. Pour la seconde partie, l'idée est qu'au maximum une donnée peut être envoyée (ou reçue) par chacune des extrémités d'une *tranche* de processeur.

Lemme 4.6. *Soit τ le temps optimal de redistribution. Alors :*

$$\tau \geqslant \max \left\{ \max_{1 \leqslant i \leqslant n} |\delta_i|, \max_{1 \leqslant i \leqslant n, 1 \leqslant l \leqslant n-1} \left\lceil \frac{|\delta_{i,i+l}|}{2} \right\rceil \right\} \times c. \tag{4.5}$$

Démonstration. Soit P_i un processeur ayant un déséquilibre positif ($\delta_i > 0$). Même si P_i peut envoyer des données à ses deux voisins, il ne peut pas le faire *simultanément* à cause de la contrainte du modèle 1-port. Le temps nécessaire pour envoyer δ_i données est donc $\delta_i \times c$ quelles que soient les destinations de ces données. Un résultat similaire est obtenu pour le cas $\delta_i < 0$. Par conséquent, une première borne inférieure du temps de redistribution est :

$$\tau \geqslant \left(\max_{1 \leqslant i \leqslant n} |\delta_i| \right) \times c.$$

Considérons une tranche non triviale de processeurs consécutifs $C_{k,l}$. Par « non triviale », nous entendons que la tranche ne peut pas être réduite à un seul processeur et qu'elle ne contient pas tous les processeurs. Nous supposons que $\delta_{k,l} > 0$. Donc, dans un schéma de redistribution quelconque, au moins $\delta_{k,l}$ données doivent être envoyées par $C_{k,l}$. Comme cette chaîne n'est pas réduite à un seul processeur, les deux processeurs des extrémités de la tranche, P_k et P_l, peuvent envoyer simultanément des données à leurs voisins à l'extérieur de la tranche, respectivement P_{k-1} et P_{l+1}. Ainsi, pendant l'intervalle de temps c, au plus deux données peuvent être envoyées à l'extérieur de la tranche. Le temps nécessaire pour que $C_{k,l}$ envoie $\delta_{k,l}$ données est donc au moins $\left\lceil \frac{\delta_{k,l}}{2} \right\rceil \times c$. Le même résultat est bien sûr obtenu dans le cas de la réception de données ($\delta_{k,l} < 0$). Par conséquent, nous obtenons la seconde borne :

$$\tau \geqslant \left(\max_{1 \leqslant i \leqslant n, 1 \leqslant l \leqslant n-1} \left\lceil \frac{|\delta_{i,i+l}|}{2} \right\rceil \right) \times c.$$

Nous regroupons alors les deux bornes précédentes afin d'obtenir la borne désirée. ■

4.5.2 Algorithme optimal

Algorithme 4.3 Algorithme de redistribution pour anneaux homogènes bidirectionnels (itération s)

Soit $\delta_{\max} = \max\{\max_{1 \leqslant i \leqslant n} |\delta_i|, \max_{1 \leqslant i \leqslant n, 1 \leqslant l \leqslant n-1} \left\lceil \frac{|\delta_{i,i+l}|}{2} \right\rceil\}$
Si $\delta_{\max} \geqslant 1$ **Alors**
  Si $\delta_{\max} \neq 2$ **Alors**
    Pour tout tranche $C_{k,l}$ telle que $\delta_{k,l} > 1$ et $\left\lceil \frac{|\delta_{k,l}|}{2} \right\rceil = \delta_{\max}$ **:**
      P_k envoie une donnée à P_{k-1} pendant l'intervalle de temps $[(s-1) \times c, s \times c[$.
      P_l envoie une donnée à P_{l+1} pendant l'intervalle de temps $[(s-1) \times c, s \times c[$.
    Pour tout tranche $C_{k,l}$ telle que $\delta_{k,l} < -1$ et $\left\lceil \frac{|\delta_{k,l}|}{2} \right\rceil = \delta_{\max}$ **:**
      P_{k-1} envoie une donnée à P_k pendant l'intervalle de temps $[(s-1) \times c, s \times c[$.
      P_{l+1} envoie une donnée à P_l pendant l'intervalle de temps $[(s-1) \times c, s \times c[$.
  Sinon Si $\delta_{\max} = 2$ **Alors**
    Pour tout tranche $C_{k,l}$ telle que $\delta_{k,l} \geqslant 3$ **:**
      P_l envoie une donnée à P_{l+1} pendant l'intervalle de temps $[(s-1) \times c, s \times c[$.
    Pour tout tranche $C_{k,l}$ telle que $\delta_{k,l} = 4$ **:**
      P_k envoie une donnée à P_{k-1} pendant l'intervalle de temps $[(s-1) \times c, s \times c[$.
    Pour tout tranche $C_{k,l}$ telle que $\delta_{k,l} \leqslant -3$ **:**
      P_{k-1} envoie une donnée à P_k pendant l'intervalle de temps $[(s-1) \times c, s \times c[$.
    Pour tout tranche $C_{k,l}$ telle que $\delta_{k,l} = -4$ **:**
      P_{l+1} envoie une donnée à P_l pendant l'intervalle de temps $[(s-1) \times c, s \times c[$.
  Pour tout processeur P_i tel que $\delta_i = \delta_{\max}$ **:**
    Si P_i n'est pas déjà en train d'envoyer une donnée pendant l'intervalle de temps $[(s-1) \times c, s \times c[$, à cause d'un des cas précédents **Alors**
      P_i envoie une donnée à P_{i+1} pendant l'intervalle de temps $[(s-1) \times c, s \times c[$.
  Pour tout processeur P_i tel que $\delta_i = -(\delta_{\max})$ **:**
    Si P_i n'est pas déjà en train de recevoir une donnée pendant l'intervalle de temps $[(s-1) \times c, s \times c[$ à cause d'un des cas précédents **Alors**
      P_i reçoit une donnée de P_{i-1} pendant l'intervalle de temps $[(s-1) \times c, s \times c[$.
  Si $\delta_{\max} = 1$ **Alors**
    Pour tout processeur P_i tel que $\delta_i = 0$ **:**
      Si P_{i-1} envoie une donnée à P_i pendant l'intervalle de temps $[(s-1) \times c, s \times c[$ **Alors**
        P_i envoie une donnée à P_{i+1} pendant l'intervalle de temps $[(s-1) \times c, s \times c[$.
      Si P_{i+1} envoie une donnée à P_i pendant l'intervalle de temps $[(s-1) \times c, s \times c[$ **Alors**
        P_i envoie une donnée à P_{i-1} pendant l'intervalle de temps $[(s-1) \times c, s \times c[$.
  Appel récursif de l'algorithme 4.3 $(s+1)$

L'algorithme 4.3 (voir ci-dessus) est un algorithme récursif qui définit des schémas de communication afin de diminuer la valeur de $\delta_{\max}$ (calculée à l'étape 1) par pas de un à chaque appel de l'algorithme. L'intuition derrière l'algorithme 4.3 est la suivante :

1. Toute tranche $C_{k,l}$ telle que $\left\lceil \frac{|\delta_{k,l}|}{2} \right\rceil = \delta_{\max}$ et $\delta_{k,l} \geqslant 0$ doit envoyer deux données par appel récursif, une par chacune de ses extrémités.
2. Toute tranche $C_{k,l}$ telle que $\left\lceil \frac{|\delta_{k,l}|}{2} \right\rceil = \delta_{\max}$ et $\delta_{k,l} \leqslant 0$ doit recevoir deux données par appel récursif, une par chacune de ses extrémités.
3. Une fois que les communications obligatoires indiquées par les deux cas précédents sont définies, nous examinons tout processeur P_i tel que $|\delta_i| = \delta_{\max}$. Si P_i est déjà impliqué dans une communication due aux cas précédents, tout est arrangé. Sinon, nous avons la liberté de qui P_i recevra une donnée (cas $\delta_i < 0$) ou à qui P_i enverra une donnée (cas $\delta_i > 0$). De manière à simplifier l'algorithme, nous décidons que toutes ces communications ont lieu dans le sens des indices croissants.

L'algorithme 4.3 est appelé la première fois avec le paramètre $s = 1$. Pour chaque appel de l'algorithme 4.3, toutes les communications ont lieu en parallèle et au même moment, puisque les chemins de communication sont homogènes par hypothèse. Un point important de l'algorithme 4.3 est que cet algorithme est un ensemble de règles indiquant *seulement* quel processeur P_i doit envoyer une donnée à quel processeur P_j, dans ses voisins immédiats. Par conséquent, quel que soit le nombre de règles décidant que des données doivent être envoyées d'un processeur P_i à son voisin immédiat P_j, il suffit de l'envoi d'une seule donnée de P_i à P_j pour satisfaire toutes ces règles.

Lemme 4.7. *L'algorithme 4.3 satisfait toutes les contraintes du modèle* 1*-port.*

Démonstration. Nous appelons *tranche maximale* une tranche $C_{k,l}$ de processeurs consécutifs dont le déséquilibre total satisfait la condition : $\lceil \frac{|\delta_{k,l}|}{2} \rceil = \delta_{\max}$. Nous appelons *processeur maximal* un processeur P_i dont le déséquilibre est égal à $\delta_{\max}$ ou $-\delta_{\max}$: $|\delta_i| = \delta_{\max}$. Les tranches maximales sont régies par les règles définies aux étapes 4 à 18, tant que les processeurs maximaux sont traités par les règles des étapes 19 et 22.

Afin de prouver que l'ensemble des règles obéit au modèle 1-port, nous prouvons qu'aucun processeur ne reçoit simultanément une donnée de ses deux voisins, et qu'aucun processeur n'envoie de données en même temps à ses deux voisins. Nous étudions seulement les cas où un processeur reçoit des données de ses deux voisins puisque l'algorithme est symétrique dans l'envoi et la réception de données.

Nous prouvons ce résultat par contradiction. Supposons qu'il existe un processeur P_j qui reçoit simultanément une donnée de chacun de ses voisins P_{j-1} et P_{j+1}. Il y a quatre cas à considérer :

1. P_{j-1} et P_{j+1} envoient tous les deux une donnée à P_j du fait des étapes 4 à 18. Alors P_{j-1} et P_{j+1} envoient une donnée à P_j soit parce qu'ils sont extrémités d'une tranche maximale positive, soit parce que P_j est extrémité d'une tranche négative. Nous avons ainsi trois sous-cas à étudier

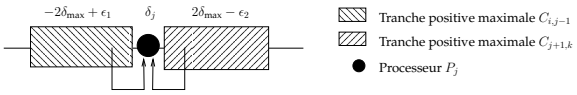


FIG. 4.2 – Cas 1a dans la preuve du lemme 4.7.

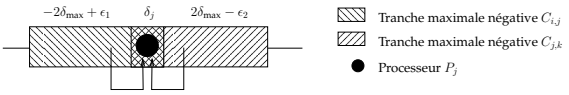


FIG. 4.3 – Cas 1b dans la preuve du lemme 4.7.

(a) P_{j-1} et P_{j+1} sont les extrémités de tranches maximales positives. Alors il existe deux indices i et j tels que les tranches $C_{i,j-1}$ et $C_{j+1,k}$ sont toutes les deux des tranches positives maximales. Donc, par définition, il existe deux valeurs ϵ_1 et ϵ_2, chacune étant égale à 0 ou 1, telles que $\delta_{i,j-1} = 2\delta_{\max} - \epsilon_1$ et $\delta_{j+1,k} = 2\delta_{\max} - \epsilon_2$. Ce cas est illustré par la figure 4.2.
Considérons la tranche $C_{i,k}$. Par définition de $\delta_{\max}$, nous avons :

$$\begin{aligned} \left\lceil \frac{\delta_{i,k}}{2} \right\rceil &\leqslant \delta_{\max} &&\Leftrightarrow \\ \left\lceil \frac{(2\delta_{\max}-\epsilon_1)+\delta_j+(2\delta_{\max}-\epsilon_2)}{2} \right\rceil &\leqslant \delta_{\max} &&\Leftrightarrow \\ 4\delta_{\max} + \delta_j - \epsilon_1 - \epsilon_2 &\leqslant 2\delta_{\max} &&\Leftrightarrow \\ \delta_j &\leqslant \epsilon_1 + \epsilon_2 - 2\delta_{\max} \end{aligned}$$

Cependant, par définition de $\delta_{\max}$, δ_j est supérieur ou égal à $-\delta_{\max}$. Donc nous arrivons à la contrainte :

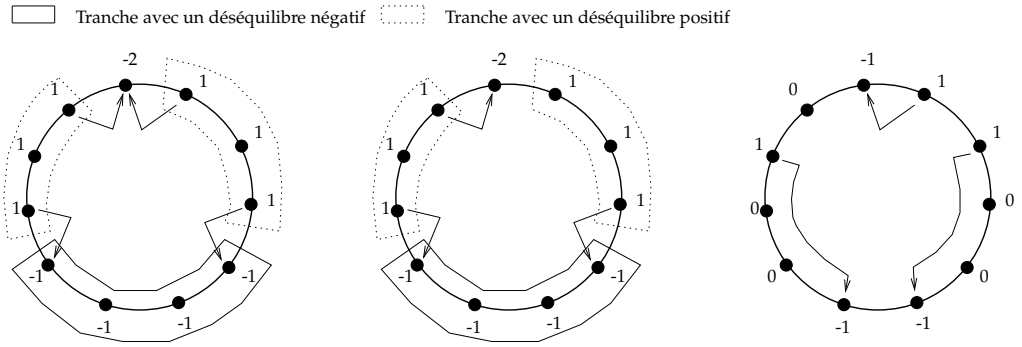
$$-\delta_{\max} \leqslant \epsilon_1 + \epsilon_2 - 2\delta_{\max} \quad \Leftrightarrow \quad \delta_{\max} \leqslant \epsilon_1 + \epsilon_2. \tag{4.6}$$

Nous avons alors trois cas à considérer :

i. $\delta_{\max} = 0$: il n'y a rien à faire, comme l'indique l'étape 2. (Dans le restant de la preuve, nous ne considérerons plus les cas où $\delta_{\max} = 0$.)

ii. $\delta_{\max} = 1$. Soit $\epsilon_1 = 1$ et $\delta_{i,j-1} = 1$, soit $\epsilon_2 = 1$ et $\delta_{j+1,k} = 1$: dans les deux cas, cela contredit notre hypothèse comme quoi P_{j-1} et P_{j+1} ont envoyé des données à P_j du fait des étapes 4 à 18.

iii. $\delta_{\max} = 2$. Ce cas est illustré à la figure 4.4. L'équation 4.6 entraîne que $\epsilon_1 = \epsilon_2 = 1$. L'application du schéma général, défini par les étapes 4 à 6 mènerait à la violation du modèle 1-port (cf. figure 4.4(a)). Cependant, chacune des deux tranches $C_{i,j-1}$ et $C_{j+1,k}$ a besoin seulement de sortir trois données lors des deux appels récursifs de l'algorithme 4.3 (les appels avec $\delta_{\max} = 2$ et $\delta_{\max} = 1$). Donc, nous demandons seulement à ces tranches de sortir une donnée pendant l'exécution de l'algorithme avec $\delta_{\max} = 2$, dans la direction P_i vers P_{i+1} (cf. figures 4.4(b) et 4.4(c)). Remarquons dans notre exemple que P_i sort une donnée à l'étape $\delta_{\max} = 2$: ce n'est pas parce qu'il est extrémité de $C_{i,j-1}$ avec $\delta_{i,j-1} = 3$ mais parce que $\delta_{i,k} = 4$.
Ce cas particulier est l'une des raisons pour lesquelles nous introduisons un traitement spécial aux étapes 10 à 18.



(a) Comportement selon la règle générale

(b) Traitement spécial : première étape

(c) Traitement spécial : seconde étape

FIG. 4.4 – Cas 1(a)iii de la preuve du lemme 4.7. La figure 4.4(a) montre le problème : le modèle 1-port est violé si nous appliquons les règles générales à ce cas. Les figures 4.4(b) et 4.4(c) décrivent les deux étapes du traitement spécial : dans la première étape, seulement une donnée est sortie par le côté droit de la tranche maximale ; et dans la seconde étape, seulement une donnée est sortie par le côté gauche de la tranche maximale.

(b) P_j est l'extrémité des deux tranches maximales négatives $C_{i,j}$ et $C_{j,k}$ avec $i < j < k$.

Donc, par définition, il existe deux valeurs ϵ_1 et ϵ_2, chacune étant égale à 0 ou 1, telle que $\delta_{i,j} = -2\delta_{\max} + \epsilon_1$ et $\delta_{j,k} = -2\delta_{\max} + \epsilon_2$. Ce cas est illustré par la figure 4.3.

Considérons la tranche $C_{i,k}$:

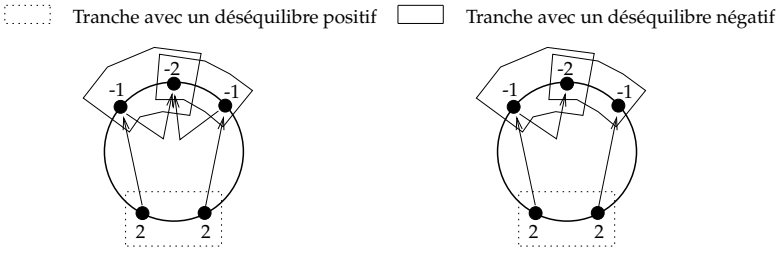
$$\delta_{i,k} = \delta_{i,j} + \delta_{j,k} - \delta_j = -4\delta_{\max} + \epsilon_1 + \epsilon_2 - \delta_j$$

Par définition de $\delta_{\max}$, nous avons : $\delta_{i,k} \geqslant -2\delta_{\max}$. Donc, $\delta_j \leqslant -2\delta_{\max} + \epsilon_1 + \epsilon_2$. Mais $\delta_j \geqslant -\delta_{\max}$. Par conséquent, $\delta_{\max} \leqslant \epsilon_1 + \epsilon_2$. Nous avons alors deux cas à considérer :

i. $\delta_{\max} = 1$. Soit $\epsilon_1 = 1$ et $\delta_{i,j} = -1$, soit $\epsilon_2 = 1$ et $\delta_{j,k} = -1$. Dans les deux cas, cela contredit notre hypothèse comme quoi P_{j-1} et P_{j+1} ont envoyé des données à P_j du fait des étapes 4 à 18.

ii. $\delta_{\max} = 2$. Alors $\epsilon_1 = \epsilon_2 = 1$ et $\delta_{i,j} = \delta_{j,k} = -3$. Comme $\delta_j \leqslant -2\delta_{\max} + \epsilon_1 + \epsilon_2$ et comme, par définition de $\delta_{\max}$, $\delta_j \geqslant -\delta_{\max}$, alors $\delta_j = -\delta_{\max} = -2$.

L'application du schéma général, défini par les étapes 4 à 6 mènerait à la violation du modèle 1-port (cf. figure 4.5(a)). Cependant, chacune des deux tranches $C_{i,j}$ et $C_{j,k}$ a besoin seulement de sortir trois données lors des deux appels récursifs restants à l'algorithme 4.3 (les appels avec $\delta_{\max} = 2$ et $\delta_{\max} = 1$). Donc, nous demandons seulement à ces tranches de sortir une donnée pendant l'exécution de l'algorithme avec $\delta_{\max} = 2$,



(a) Comportement selon la règle générale

(b) Traitement spécial : première étape

(c) Traitement spécial : seconde étape

FIG. 4.5 – Cas 1(b)ii de la preuve du lemme 4.7. La figure 4.5(a) montre le problème : le modèle 1-port est violé si nous appliquons les règles générales à ce cas. Les figures 4.5(b) et 4.5(c) décrivent les deux étapes du traitement spécial : dans la première étape, seulement une donnée est entrée par le côté droit de la tranche maximale ; et dans la seconde étape, seulement une donnée est entrée par le côté gauche de la tranche maximale.

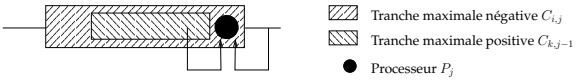


FIG. 4.6 – Cas 1(c)i de la preuve du lemme 4.7.

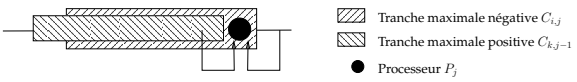


FIG. 4.7 – Cas 1(c)ii de la preuve du lemme 4.7.

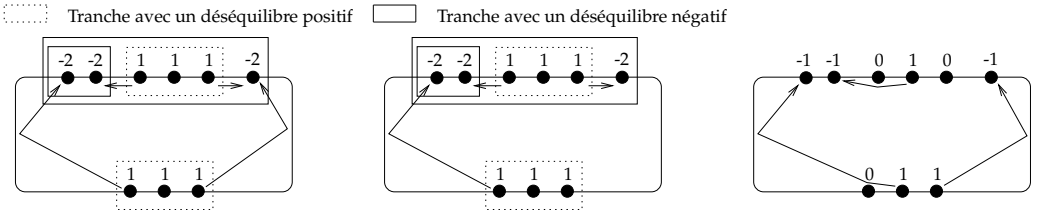
dans la direction P_i vers P_{i+1} (cf. figures 4.5(b) et 4.5(c)). Remarquons dans notre exemple que P_i sort une donnée à l'étape $\delta_{\max} = 2$: ce n'est pas parce qu'il est extrémité de $C_{i,j-1}$ avec $\delta_{i,j-1} = -3$ mais parce que $\delta_{i,k} = -4$.

Ce cas particulier est l'une des raisons pour lesquelles nous introduisons un traitement spécial aux étapes 10 à 18.

(c) P_j est l'extrémité d'une tranche maximale négative et l'un de ses voisins est l'extrémité d'une tranche maximale positive. Sans perte de généralité, supposons que P_{j+1} envoie une donnée à P_j car $C_{i,j}$ est une tranche maximale négative. Alors, P_{j-1} envoie une donnée à P_j parce qu'il est extrémité d'une tranche maximale positive $C_{k,j-1}$. Donc, par définition, il existe deux valeurs ϵ_1 et ϵ_2, chacune étant égale à 0 ou 1, telle que $\delta_{i,j} = -2\delta_{\max} + \epsilon_1$ et $\delta_{k,j-1} = 2\delta_{\max} - \epsilon_2$. Nous avons deux cas à considérer, dépendant de si la tranche $C_{k,j-1}$ est comprise dans la tranche $C_{i,j}$ ou non :

i. $k \in [i, j-2]$ (ce cas est illustré par la figure 4.6). $\delta_{i,k-1} + \delta_j = \delta_{i,j} - \delta_{k,j-1} = (-2\delta_{\max} + \epsilon_1) - (2\delta_{\max} - \epsilon_2) = -4\delta_{\max} + \epsilon_1 + \epsilon_2$. Cependant, par définition de $\delta_{\max}$, $\delta_{i,k-1} \geqslant -2\delta_{\max}$ et $\delta_j \geqslant -\delta_{\max}$. Donc, $\delta_{\max} \leqslant \epsilon_1 + \epsilon_2$. Une nouvelle fois, nous avons deux cas à considérer :

A. $\delta_{\max} = 1$. Alors, comme toujours, soit $\epsilon_1 = 1$ et $\delta_{i,j} = -1$, soit $\epsilon_2 = 1$



(a) Comportement selon la règle générale

(b) Traitement spécial : première étape

(c) Traitement spécial : seconde étape

FIG. 4.8 – Cas 1(c)iB de la preuve du lemme 4.7. La figure 4.8(a) montre le problème : le modèle 1-port est violé si nous appliquons les règles générales à ce cas. Les figures 4.8(b) et 4.8(c) décrivent les deux étapes du traitement spécial : dans la première étape, seulement une donnée est sortie par le côté droit de la tranche maximale ; et dans la seconde étape, seulement une donnée est sortie par le côté gauche de la tranche maximale.

et $\delta_{j,k} = 1$. Dans les deux cas, cela contredit notre hypothèse sur $C_{i,j}$ et $C_{k,j-1}$.

B. $\delta_{\max} = 2$. Alors $\epsilon_1 = \epsilon_2 = 1$, $\delta_{i,j} = -3$ et $\delta_{k,j-1} = 3$. Ainsi, $\delta_{i,k-1} + \delta_j = -6$. Par définition de $\delta_{\max}$, $\delta_j \geqslant -\delta_{\max} = -2$ et $\delta_{i,k-1} \geqslant -2\delta_{\max} = -4$, nous avons $\delta_j = -2$ et $\delta_{i,k-1} = -4$.

De façon similaire aux cas 1(a)iii et 1(b)ii, l'application du schéma général défini des étapes 4 à 9 mènerait à une violation du modèle 1-port (cf. figure 4.8(a)). Cependant, la tranche $C_{i,j}$ a besoin seulement de sortir trois données lors des deux appels récursifs de l'algorithme 4.3 (les appels avec $\delta_{\max} = 2$ et $\delta_{\max} = 1$). Donc, nous demandons seulement à la tranche $C_{i,j}$ de faire entrer une donnée et à la tranche $C_{k,j-1}$ de faire sortir une donnée pendant l'exécution de l'algorithme avec $\delta_{\max} = 2$, les communications étant dans la direction P_i vers P_{i+1} (cf. figures 4.8(b) et 4.8(c)). Remarquons dans notre exemple que P_k sort une donnée à l'étape $\delta_{\max} = 2$: ce n'est pas parce qu'il est extrémité de $C_{k,j-1}$ avec $\delta_{k,j-1} = 3$ mais parce que $\delta_{i,k-1} = -4$.

Ce cas particulier est l'une des raisons pour lesquelles nous introduisons un traitement spécial aux étapes 10 à 18.

ii. $k < i$ (ce cas est illustré par la figure 4.7). Alors, $\delta_{k,i-1} = \delta_j + \delta_{k,j-1} - \delta_{i,j} = \delta_j + (2\delta_{\max} - \epsilon_2) - (2\delta_{\max} + \epsilon_1) = \delta_j + 4\delta_{\max} - \epsilon_1 - \epsilon_2$. Cependant, par définition de $\delta_{\max}$, $\delta_{k,i-1} \leqslant 2\delta_{\max}$ et $\delta_j \geqslant -\delta_{\max}$. Donc, $\delta_j \leqslant -2\delta_{\max} - \epsilon_1 - \epsilon_2$ et, donc, $-\delta_{\max} \leqslant -2\delta_{\max} - \epsilon_1 - \epsilon_2$. Par conséquent, $\delta_{\max} = \epsilon_1 = \epsilon_2 = 0$, ce qui est absurde.

2. P_{j-1} et P_{j+1} sont les deux processeurs qui envoient des données à P_j : l'un en-

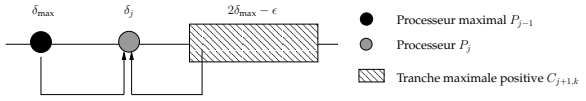


FIG. 4.9 – Cas 2a de la preuve du lemme 4.7.

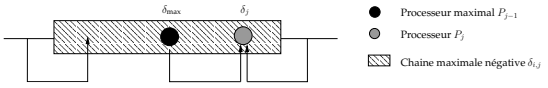


FIG. 4.10 – Cas 2b de la preuve du lemme 4.7.

voie des données à cause des étapes 4 à 18 ; l'autre est un processeur maximal envoyant des données à cause des étapes 19 et 24. Sans perte de généralité, supposons que P_{j-1} est le processeur maximal.
Nous avons deux cas à considérer, à savoir si P_{j+1} envoie une donnée à P_j à cause d'une tranche maximale positive ou négative.

(a) P_{j+1} est l'extrémité d'une tranche maximale positive $C_{j+1,k}$. La figure 4.9 illustre ce cas. Ainsi, il existe $\epsilon \in \{0;1\}$, tel que $\delta_{j+1,k} = 2\delta_{\max} - \epsilon$. Par hypothèse, P_{j-1} envoie des données à cause des étapes 19 et 24, et donc la tranche $C_{j-1,k}$ n'est pas une tranche maximale, c'est-à-dire, $\lceil \frac{\delta_{j-1,k}}{2} \rceil \leqslant \delta_{\max} - 1$.

$$\begin{aligned} \left\lceil \frac{\delta_{j-1,k}}{2} \right\rceil &= \left\lceil \frac{\delta_{\max} + \delta_j + 2\delta_{\max} - \epsilon}{2} \right\rceil \leqslant \delta_{\max} - 1 \\ \Leftrightarrow \quad & \delta_{\max} + \delta_j + 2\delta_{\max} - \epsilon \leqslant 2\delta_{\max} - 2 \\ \Leftrightarrow \quad & \delta_j \leqslant \epsilon - 2 - \delta_{\max} \\ \Rightarrow \quad & \delta_j \leqslant -1 - \delta_{\max} \end{aligned}$$

ce qui contredit la définition de $\delta_{\max}$.

(b) P_j est l'extrémité d'une tranche maximale négative $C_{i,j}$ (la figure 4.10 illustre ce cas). Alors il existe $\epsilon \in \{0;1\}$, tel que $\delta_{i,j} = -2\delta_{\max} + \epsilon$. Ainsi, $\delta_{i,j-2} = \delta_{i,j} - \delta_{\max} - \delta_j = -3\delta_{\max} + \epsilon - \delta_j$. Par hypothèse, P_{j-1} envoie des données à cause des étapes 19 et 24, et donc, la tranche $C_{i,j-2}$ n'est pas une tranche maximale négative, ce qui implique que $\delta_{i,j-2} \geqslant -2\delta_{\max} + 2$. Ainsi, $-3\delta_{\max} + \epsilon - \delta_j \geqslant -2\delta_{\max} + 2$ et donc $\delta_j \leqslant -\delta_{\max} + \epsilon - 2 \leqslant -\delta_{\max} - 1$, ce qui contredit la définition de $\delta_{\max}$.

3. P_{j-1} et P_{j+1} sont les deux processeurs qui envoient des données à P_j car ils sont tous les deux des processeurs maximaux envoyant des données du fait des étapes 19 à 24. Ce cas est impossible car ces étapes définissent seulement l'envoi de donnée dans le sens P_i à P_{i+1} et jamais dans le sens inverse (de P_i à P_{i-1}).

4. P_j est un processeur maximal de déséquilibre négatif et c'est la raison pour laquelle P_{j-1} envoie une donnée (suivant les étapes 19 à 24). Il y a plusieurs raisons pour lesquelles P_{j+1} enverrait une donnée à P_j :

   (a) P_{j+1} est extrémité d'une tranche maximale positive $C_{j+1,k}$ et il envoie une donnée à cause des étapes 4 à 18. Alors, le test à l'étape 23 contredit notre hypothèse sur P_{j-1}.

   (b) P_{j+1} est un processeur maximal positif. Mais dans ce cas, P_{j+1} envoie une donnée à P_{j+2} et pas à P_j.

(c) P_j est extrémité d'une tranche maximale négative $C_{i,j}$ et P_{j+1} lui envoie une donnée à cause des étapes 4 à 18. Alors le test à l'étape 23 contredit notre hypothèse sur P_j.

■

Lemme 4.8. *L'algorithme 4.3 se termine en exactement*

$$\max\left\{\max_{1\leqslant i\leqslant n}|\delta_i|,\max_{1\leqslant i\leqslant n,1\leqslant l\leqslant n-1}\left\lceil\frac{\delta_{i,i+l}}{2}\right\rceil\right\}$$

appels récursifs.

Démonstration. Nous prouvons d'un appel récursif à l'autre de l'algorithme 4.3 que la valeur de $\delta_{\max}$ (calculée à l'étape 1) décroît par pas de un. Par conséquent, nous considérons la façon dont les déséquilibres changent entre l'appel initial de l'algorithme 4.3 et l'appel récursif (exclu). Pour le cas général, nous avons quatre propriétés à prouver :

1. Si la tranche $C_{k,l}$ non-triviale était initialement une tranche maximale, c'est-à-dire, si $\lceil\frac{|\delta_{k,l}|}{2}\rceil=\delta_{\max}$, alors après les communications nous avons : $\lceil\frac{|\delta_{k,l}|}{2}\rceil=\delta_{\max}-1$.

   Comme précédemment, nous nous focalisons sur le cas d'une tranche maximale positive. Les règles de l'algorithme 4.3 sont écrites de sorte que la tranche $C_{k,l}$ envoie deux données (ou seulement une dans le cas dégénéré $\delta_{\max}=2$ et $\delta_{k,l}=3$) pendant l'exécution de l'algorithme 4.3. C'est tout ce qu'il nous faut conclure, à condition que cette tranche ne reçoive pas de données pendant cet appel.

   Donc, supposons que $C_{k,l}$ reçoit une donnée. Nous avons trois cas à considérer :

   (a) La tranche maximale $C_{k,l}$ reçoit une donnée du processeur qui est extrémité d'une autre tranche maximale et qui envoie une donnée à cause des étapes 4 à 18. Comme l'autre tranche maximale envoie une donnée, son déséquilibre est positif. Sans perte de généralité, nous supposons que c'est une tranche maximale de la forme $C_{l+1,m}$. Alors, par définition des tranches maximales, $\delta_{k,l}=2\delta_{\max}-\epsilon_1$ et $\delta_{l+1,m}=2\delta_{\max}-\epsilon_2$, avec ϵ_1 et ϵ_2 étant dans $\{0,1\}$. Donc, $\delta_{k,m}=4\delta_{\max}-\epsilon_1-\epsilon_2$. Cependant, par définition de $\delta_{\max}$, $\delta_{k,m}\leqslant 2\delta_{\max}$. Par conséquent, nous obtenons $2\delta_{\max}\leqslant\epsilon_1+\epsilon_2$, ce qui implique que $\delta_{\max}=1$ et $\epsilon_1=\epsilon_2=1$. Alors $\delta_{l+1,m}=1$ ce qui contredit l'hypothèse comme quoi $C_{l+1,m}$ envoie une donnée du fait des étapes 4 à18 (voir le test à l'étape 4).

   (b) La tranche maximale $C_{k,l}$ reçoit une donnée d'un processeur qui est maximal et qui envoie une donnée du fait des étapes 19 à 24. Ce cas peut seulement arriver si ce processeur maximal a un déséquilibre positif. Sans perte de généralité, nous supposons que P_{k-1} a un déséquilibre $\delta_{\max}$. Alors, par définition de tranches maximales, $\delta_{k,l}\geqslant 2\delta_{\max}-1$ et $\delta_{k-1,l}\geqslant 3\delta_{\max}-1$. Cependant, par définition de $\delta_{\max}$, $\delta_{k-1,l}\leqslant 2\delta_{\max}$. Donc $\delta_{\max}=1$ et $\delta_{k-1}=1$. Alors nous avons $\delta_{k,l}=1$, et $\delta_{k-1,l}=2$. Par conséquent, $C_{k-1,l}$ est une tranche maximale et le processeur P_{k-1} envoie une donnée au processeur P_{k-2} plutôt qu'au processeur P_k.

(c) La tranche maximale $C_{k,l}$ reçoit une donnée parce que l'une de ses extrémités est aussi extrémité d'une tranche maximale négative. Sans perte de généralité, nous supposons que la tranche maximale négative est de la forme $C_{k,m}$ (avec $l \in [k;m]$). $C_{k,l}$ étant une tranche maximale positive, $\delta_{k,l} = 2\delta_{\max} - \epsilon_1$ avec $\epsilon_1 \in \{0;1\}$. $C_{k,m}$ étant une tranche maximale négative, $\delta_{k,m} = -2\delta_{\max} + \epsilon_2$ avec $\epsilon_2 \in \{0;1\}$. Nous avons deux cas à considérer :

i. $l < m$. alors, $\delta_{l+1,m} = (-2\delta_{\max} + \epsilon_2) - (2\delta_{\max} - \epsilon_1) = -4\delta_{\max} + \epsilon_1 + \epsilon_2$. Par définition de $\delta_{\max}$, $\delta_{l+1,m} \geqslant -2\delta_{\max}$, et donc $\delta_{\max} = 1$ et $\epsilon_1 = \epsilon_2 = 1$.

ii. $l > m$. Alors $\delta_{m+1,l} = (2\delta_{\max} - \epsilon_1) - (-2\delta_{\max} + \epsilon_2) = 4\delta_{\max} - \epsilon_1 - \epsilon_2$. Par définition de $\delta_{\max}$, $\delta_{m+1,l} \leqslant 2\delta_{\max}$, et donc $\delta_{\max} = 1$ et $\epsilon_1 = \epsilon_2 = 1$. D'où $C_{k,m} = -1$.

Donc dans les deux cas, $C_{k,m} = -1$. D'où, à cause du test de l'étape 7, les règles des étapes 8 et 9 ne s'appliquent pas, et la tranche maximale $C_{k,l}$ ne reçoit pas de donnée car l'une de ses extrémités est déjà extrémité d'une tranche maximale négative.

2. Si le processeur P_i était initialement maximal, c'est-à-dire, si $|\delta_i| = \delta_{\max}$, alors, après communications, nous avons $|\delta_i| = \delta_{\max} - 1$.

   Comme précédemment, nous nous focalisons sur le cas $\delta_i = \delta_{\max}$. Si après communications, nous n'avons pas $|\delta_i| = \delta_{\max} - 1$, alors P_i a reçu une donnée.

   (a) P_i reçoit une donnée d'un processeur qui est l'extrémité d'une tranche maximale et qui envoie une donnée du fait des étapes 19 à 24. Sans perte de généralité, nous supposons que ce processeur est P_{i+1} et la tranche $C_{i+1,j}$. Par définition des tranches maximales, il existe une valeur ϵ, égale à 0 ou 1 telle que $\delta_{i+1,j} = 2\delta_{\max} - \epsilon$. D'où $\delta_{i,j} = 3\delta_{\max} - \epsilon$. Comme par définition de $\delta_{\max}$, $\delta_{i,j} \leqslant 2\delta_{\max}$, cela mène à $\delta_{\max} = \epsilon = 1$. Donc, $\delta_{i+1,j} = 1$, ce qui contredit notre hypothèse sur P_{i+1}.

   (b) P_i reçoit une donnée, du fait des étapes 19 à 24, puisqu'il est extrémité d'une tranche maximale négative. Sans perte de généralité, supposons que la tranche est $C_{i,j}$. Par définition des tranches maximales, il existe une valeur ϵ, égale à 0 ou 1 telle que $\delta_{i,j} = -2\delta_{\max} + \epsilon$. Alors $\delta_{i+1,j} = (-2\delta_{\max} + \epsilon) - \delta_{\max} = -3\delta_{\max} + \epsilon$. Comme par définition de $\delta_{\max}$, $\delta_{i+1,j} \geqslant -2\delta_{\max}$, cela mène à $\delta_{\max} = \epsilon = 1$. Donc $\delta_{i,j} = -1$, ce qui contredit notre hypothèse sur P_i.

   (c) P_i reçoit une donnée d'un autre processeur maximal, disons P_{i-1}, qui envoie des données du fait des étapes 19 et 24. Or, deux processeurs maximaux côte à côte définissent une tranche maximale. Par conséquent une contradiction puisque dans une tranche maximale $\delta_{i-1,i}$, le processeur P_{i-1} envoie une donnée au processeur P_{i-2} et pas à P_i.

3. Après que les communications aient eu lieu, aucun processeur P_i est tel que $|\delta_i| = \delta_{\max}$.

   Comme précédemment, considérons le cas $\delta_i = \delta_{\max}$ après que les communications aient eu lieu. À cause du cas 2, un tel cas se présenterait seulement si le

déséquilibre de P_i était égal à $\delta_{\max} - 1$ avant les communications (en raison du modèle 1-port garanti par le lemme 4.7) et si P_i recevait une donnée mais n'en envoyait aucune.

Nous avons trois cas à considérer :

(a) Le processeur P_i reçoit une donnée d'un processeur qui est extrémité d'une tranche maximale et qui envoie des données en raison des étapes 4 à 18. Il n'existe pas de configuration dans laquelle la tranche maximale est négative. Donc, la tranche maximale est positive. Sans perte de généralité, supposons que c'est une tranche maximale de la forme $C_{i+1,j}$. Alors, par définition des tranches maximales, $\delta_{i+1,j} = 2\delta_{\max} - \epsilon_1$ et ϵ_1 est égal à 0 ou 1. Donc, $\delta_{i,j} = 3\delta_{\max} - \epsilon_1 - 1$. Cependant, $C_{i,j}$ n'est pas une tranche maximale (par hypothèse, P_i n'envoie pas de données). Ainsi, par définition de $\delta_{\max}$, $\delta_{i,j} \leqslant 2\delta_{\max} - 2$. Par conséquent, nous obtenons $\delta_{\max} \leqslant \epsilon_1 - 1$ qui n'a aucune solution.

(b) Le processeur P_i reçoit une donnée d'un processeur qui est maximal et qui envoie des données en raison des étapes 4 à 18. Ce cas arrive seulement si ce processeur maximal a un déséquilibre positif. Sans perte de généralité, supposons que le processeur P_{i-1} a un déséquilibre de $\delta_{\max}$. Alors, $\delta_{i-1,i} = 2\delta_{\max} - 1$. Donc, $\delta_{i-1,i}$ est une tranche maximale, ce qui contredit notre hypothèse sur P_{i-1}.

(c) Le processeur P_i reçoit une donnée puisqu'il est extrémité d'une tranche maximale négative, $C_{i,j}$ par exemple. Alors, par définition de tranches maximales, il existe $\epsilon \in \{0, 1\}$ tel que $\delta_{i,j} = -2\delta_{\max} + \epsilon$. Par définition de $\delta_{\max}$ nous avons $\delta_{i+1,j} \geqslant -2\delta_{\max}$. Comme, $\delta_{i+1,j} = -3\delta_{\max} + \epsilon$, nous obtenons $\delta_{\max} \leqslant \epsilon$. D'où $C_{i,j} = -1$ et en raison du test 7 , les règles des étapes 8 et 9 ne s'appliquent pas, et P_i ne reçoit pas de donnée parce qu'il est extrémité d'une tranche maximale négative.

4. Après que les communications ont eu lieu, aucune tranche non-triviale $C_{k,l}$ est telle que $\lceil \frac{|\delta_{k,l}|}{2} \rceil = \delta_{\max}$.

Une nouvelle fois, considérons le cas des tranches positives. Nous pouvons supposer que la tranche $C_{k,l}$ n'était pas initialement une tranche maximale comme dans le cas déjà traité. Donc, il existe une valeur $\epsilon_1 \in \{0, 1\}$ telle que $\delta_{k,l} = 2\delta_{\max} - 2 - \epsilon_1$ et nous avons trois cas à considérer :

(a) La tranche $C_{k,l}$ reçoit une donnée d'un processeur qui est extrémité d'une tranche maximale qui envoie des données en raison des étapes 4 à 18. Il n'existe aucune configuration dans laquelle la tranche maximale est négative. Sans perte de généralité, supposons qu'elle est de la forme $C_{j,k-1}$. Alors, par définition des tranches maximales, $\delta_{j,k-1} = 2\delta_{\max} - \epsilon_2$, avec ϵ_2 dans $\{0, 1\}$. Donc, $\delta_{j,l} = 4\delta_{\max} - \epsilon_1 - \epsilon_2 - 2$. Cependant, par définition de $\delta_{\max}$, $\delta_{j,l} \leqslant 2\delta_{\max}$. Par conséquent, nous obtenons $2\delta_{\max} \leqslant \epsilon_1 + \epsilon_2 + 2$. Nous avons deux sous-cas à considérer :

i. $\delta_{\max} = 2$. Alors, $\epsilon_1 = \epsilon_2 = 1$. Cependant, dans le cas $\delta_{j,l} = 4$, $C_{j,l}$ est une tranche maximale, et P_l envoie une donnée à P_{l+1}. Avant que les

communications aient lieu, $\delta_{k,l} = 1$. Lors des communications $C_{k,l}$ reçoit au plus deux données (puisqu'elle a deux extrémités) et en envoie au moins une à partir de P_l. Donc, après que les communications aient eu lieu, $\delta_{k,l}$ est égal à 0, 1 ou 2, et les trois cas sont bons.

ii. $\delta_{\max} = 1$. Nous pouvons alors conclure en utilisant les résultats des cas 2 et 3.

(b) La tranche $C_{k,l}$ reçoit une donnée car elle est encapsulée dans une tranche maximale négative. Sans perte de généralité, supposons que cette tranche maximale négative est de la forme $C_{k,m}$. $C_{k,m}$ étant une tranche maximale négative, $\delta_{k,m} = -2\delta_{\max} + \epsilon_2$ avec $\epsilon_2 \in \{0; 1\}$.

i. $l < m$. Alors $\delta_{l+1,m} = (-2\delta_{\max} + \epsilon_2) - (2\delta_{\max} - 2 - \epsilon_1) = -4\delta_{\max} + \epsilon_1 + \epsilon_2 + 2$. Par définition de $\delta_{\max}$, $\delta_{l+1,m} \geqslant -2\delta_{\max}$. Le cas $\delta_{\max} = 1$ est résolu en utilisant le résultat du cas 3. Alors $\delta_{\max} = 2$, $\epsilon_1 = \epsilon_2 = 1$, $\delta_{k,l} = 1$ et $\delta_{l+1,m} = -4$. Donc, $C_{l+1,m}$ est une tranche maximale négative et P_l envoie une donnée à P_{l+1}. Donc le déséquilibre de $C_{k,l}$, qui était à l'origine égale à 1 augmente tout au plus d'un entre avant et après que les communications aient eu lieu, et il n'y a aucun problème.

ii. $m < l$. Alors $\delta_{m+1,l} = (2\delta_{\max} - 2 - \epsilon_1) - (-2\delta_{\max} + \epsilon_2) = 4\delta_{\max} - \epsilon_1 - \epsilon_2 - 2$. Par définition de $\delta_{\max}$, $\delta_{m+1,l} \leqslant 2\delta_{\max}$. Le cas $\delta_{\max} = 1$ est résolu en utilisant le résultat du cas 3. Alors $\delta_{\max} = 2$, $\epsilon_1 = \epsilon_2 = 1$, $\delta_{k,l} = 1$ et $\delta_{m+1,l} = -4$. Donc, $C_{m+1,l}$ est une tranche maximale négative et P_m envoie des données à P_{m+1}. Donc le déséquilibre de $C_{k,l}$, qui était à l'origine égal à 1 augmente tout au plus d'un entre avant et après que les communications aient eu lieu, et il n'y a aucun problème.

(c) La tranche $C_{k,l}$ reçoit seulement une donnée, et d'un processeur qui est maximal et qui envoie des données en raison des étapes 19 à 24. Ce cas peut se produire seulement si ce processeur maximal a un déséquilibre positif. Alors, par l'étape 19, c'est le processeur P_{k-1} qui a un déséquilibre de $\delta_{\max}$. Pour que la tranche $C_{k,l}$ soit telle que $\lceil \frac{|\delta_{k,l}|}{2} \rceil = \delta_{\max}$ après que les communications aient eu lieu, nécessairement, $\delta_{k,l} \geqslant 2\delta_{\max} - 2$ avant les communications. Alors $\delta_{k-1,l} \geqslant 3\delta_{\max} - 2$. Comme nous avons supposé que P_{i-1} envoie des données à cause des étapes 19 à 24, la tranche $C_{k-1,l}$ n'est pas maximale et donc $\delta_{k-1,l} \leqslant 2\delta_{\max} - 2$. Par conséquent $\delta_{\max} \leqslant 0$, une contradiction.

■

L'optimalité de l'algorithme 4.3 est alors un simple corollaire du lemme 4.8 et de la borne inférieure définie par l'équation 4.5.

Théorème 4.3. *L'algorithme 4.3 est optimal.*

4.6 Anneau hétérogène bidirectionnel

Dans cette section, nous considérons le cas le plus général, c'est-à-dire l'anneau hétérogène bidirectionnel. Nous ne connaissons pas d'algorithme optimal dans ce cas de figure. Cependant, si nous supposons que chaque processeur possède initialement plus de données que ce qu'il doit envoyer pendant l'exécution de l'algorithme (ce que nous appelons une redistribution *légère*), alors nous réussissons à obtenir une solution optimale.

4.6.1 Redistribution légère

Dans toute cette section, nous supposons que nous sommes dans le cas de figure d'une redistribution *légère* : le nombre de données envoyées par n'importe quel processeur dans tout l'algorithme de redistribution est inférieur ou égal à sa charge originale. Il existe deux raisons pour qu'un processeur P_i envoie une donnée : (i) il est surchargé ($\delta_i > 0$) ; (ii) il propage des données à un autre processeur placé plus loin dans l'anneau. Si P_i possède au départ au moins autant de données qu'il devra en envoyer pendant l'exécution de l'algorithme, alors P_i peut envoyer immédiatement tout ce qu'il doit envoyer. Sinon, dans le cas général, certains processeurs doivent attendre de recevoir des données d'un voisin avant de pouvoir les retransmettre à un autre voisin.

4.6.1.1 Solution par programmation linéaire en entier

Sous l'hypothèse de « redistribution légère », nous pouvons construire un programme linéaire en entiers pour résoudre notre problème (cf. système 4.7). Soit $\mathcal{S}$ une solution et $\mathcal{S}_{i,i+1}$ le nombre de données qu'un processeur P_i envoie au processeur P_{i+1}. De la même manière, $\mathcal{S}_{i,i-1}$ est le nombre de données que P_i envoie à P_{i-1}. Afin d'alléger l'écriture des équations, nous imposons dans les deux premières équations du système 4.7 que $\mathcal{S}_{i,i+1}$ et $\mathcal{S}_{i,i-1}$ sont positifs pour tout i, ce qui impose d'utiliser d'autres variables $\mathcal{S}_{i+1,i}$ et $\mathcal{S}_{i-1,i}$ pour les communications symétriques. La troisième équation établit qu'après redistribution, il n'y a plus de déséquilibre. Nous notons τ le temps d'exécution de la redistribution. Pour tout processeur P_i, du fait du modèle 1-port, τ doit être plus grand que le temps passé par P_i à envoyer des données (quatrième équation) ou passé par P_i à recevoir des données (cinquième équation). Notre but est de minimiser τ, d'où le système :

$$\text{MINIMISER } \tau \text{ AVEC LES CONTRAINTES SUIVANTES}$$
$$\begin{cases} \mathcal{S}_{i,i+1} \geqslant 0 & 1 \leqslant i \leqslant n \\ \mathcal{S}_{i,i-1} \geqslant 0 & 1 \leqslant i \leqslant n \\ \mathcal{S}_{i,i+1} + \mathcal{S}_{i,i-1} - \mathcal{S}_{i+1,i} - \mathcal{S}_{i-1,i} = \delta_i & 1 \leqslant i \leqslant n \\ \mathcal{S}_{i,i+1} c_{i,i+1} + \mathcal{S}_{i,i-1} c_{i,i-1} \leqslant \tau & 1 \leqslant i \leqslant n \\ \mathcal{S}_{i+1,i} c_{i+1,i} + \mathcal{S}_{i-1,i} c_{i-1,i} \leqslant \tau & 1 \leqslant i \leqslant n \end{cases} \tag{4.7}$$

Lemme 4.9. *N'importe quelle solution optimale du système 4.7 est réalisable, par exemple en utilisant le schéma suivant : pour tout $i \in [1, n]$, P_i commence à envoyer ses données à P_{i+1} au temps 0 et, après la fin de cette communication, commence à envoyer ses données à P_{i-1} dès que possible sous les contraintes du modèle 1-port.*

Démonstration. Nous avons à montrer que nous sommes capable d'ordonnancer les communications définies par toute solution optimale $(\mathcal{S}, \tau)$ du système 4.7 telle que la redistribution ne prenne pas un temps plus grand que τ. Pour tout $i \in [1, n]$, nous ordonnançons au temps 0 toutes les communications de P_i vers P_{i+1}. Cette communication est faite en un temps $\mathcal{S}_{i,i+1}c_{i,i+1}$: du fait de l'hypothèse de « redistribution légère », P_i possède déjà les données qu'il doit envoyer. Grâce à la quatrième équation du système 4.7, cette communication se termine avant le temps τ.

Pour toute valeur de $i \in [1, n]$, nous devons encore ordonnancer l'envoi des données de P_i à P_{i-1}. Nous ordonnançons cette communication dès que possible, donc au temps $\max\{\mathcal{S}_{i,i+1}c_{i,i+1}, \mathcal{S}_{i-2,i-1}c_{i-2,i-1}\}$, c'est-à-dire au temps le plus tôt auquel (i) P_i a fini d'envoyer ses données à P_{i+1}, et (ii) P_{i-1} a fini de recevoir des données de P_{i-2}. La communication entre P_i et P_{i-1} se termine à la date :

$$\max\{\mathcal{S}_{i,i+1}c_{i,i+1}, \mathcal{S}_{i-2,i-1}c_{i-2,i-1}\} + \mathcal{S}_{i,i-1}c_{i,i-1} = \max\{\mathcal{S}_{i,i+1}c_{i,i+1} + \mathcal{S}_{i,i-1}c_{i,i-1}, \mathcal{S}_{i-2,i-1}c_{i-2,i-1} + \mathcal{S}_{i,i-1}c_{i,i-1}\}. \quad (4.8)$$

De nouveau, c'est vrai grâce à l'hypothèse de « redistribution légère » : aucun processeur n'a besoin d'attendre pour recevoir des données avant de pouvoir en envoyer à ses voisins.

Le premier terme de l'expression « max » est le temps nécessaire à P_i pour envoyer des données à ses deux voisins P_{i+1} et P_{i-1}. Ce terme est inférieur ou égal à τ du fait de la quatrième équation du système 4.7. Le deuxième terme de l'expression « max » est quant à lui le temps nécessaire à P_{i-1} pour recevoir des données de P_{i-2} et P_i. Ce terme est inférieur ou égal à τ à cause de la cinquième équation du système 4.7. ■

4.6.1.2 Solution par programmation linéaire en rationnel

Nous utilisons le système 4.7 pour trouver une solution optimale au problème. Si, dans cette solution optimale, pour tout processeur P_i, le nombre total de données envoyées est inférieur ou égal à la charge initiale ($\mathcal{S}_{i,i+1} + \mathcal{S}_{i,i-1} \leqslant L_i$), nous sommes sous l'hypothèse de « redistribution légère » et nous pouvons utiliser la solution du système 4.7 sans risque. Mais même si l'hypothèse de « redistribution légère » est vérifiée, on peut souhaiter résoudre le problème de redistribution avec une technique moins coûteuse que la programmation linéaire en nombres entiers (qui est exponentielle dans le pire cas). Une idée serait tout d'abord de résoudre le système 4.7 pour trouver une solution optimale *rationnelle*, ce qui se fait toujours en temps polynomial, et d'arrondir cette solution pour obtenir une bonne solution entière. En fait, le lemme suivant montre que l'une des deux solutions rationnelles mène toujours à la solution

(entière) optimale. Le problème de redistribution légère est donc de complexité polynomiale.

Proposition 4.1. *Soit $\mathcal{R}$ une solution optimale rationnelle du problème de redistribution. Pour tout j dans $[1,n]$, $\mathcal{R}_j$ représente le nombre de données que le processeur P_j envoie au processeur P_{j+1} (en utilisant les notations du système 4.7, $\mathcal{R}_j = \mathcal{S}_{j,j+1} - \mathcal{S}_{j+1,j}$). Soit $\mathcal{F}$ la solution entière définie par $\mathcal{F}_1 = \lfloor \mathcal{R}_1 \rfloor$. Soit $\mathcal{G}$ la solution entière définie par $\mathcal{G}_1 = \lceil \mathcal{R}_1 \rceil$. Alors :*
(i) $\mathcal{F}$ et $\mathcal{G}$ sont bien définies par la condition ci-dessus,
(ii) $\mathcal{F}$ ou $\mathcal{G}$ est une solution entière optimale.

Démonstration. Le lemme 4.10 ci-dessous établit que $\mathcal{F}$ et $\mathcal{G}$ sont totalement définies. Le lemme 4.11 établit qu'il existe au moins une solution entière optimale $\mathcal{E}$ telle que $|\mathcal{E}_1 - \mathcal{R}_1| < 1$. Les deux seules solutions satisfaisant ces contraintes sont $\mathcal{F}$ et $\mathcal{G}$. D'où le résultat. ■

Lemme 4.10. *Pour définir totalement le nombre de données envoyées entre les processeurs dans n'importe quel schéma de redistribution, nous avons besoin seulement de définir, pour une valeur donnée, le nombre de données que le processeur P_j envoie au processeur P_{j+1}.*

Démonstration. Sans perte de généralité, nous supposons que nous avons une valeur fixée pour $\mathcal{R}_1$, le nombre de données envoyées par P_1 à P_2. (Notons que $\mathcal{R}_1$ peut être négatif, ce qui signifie que P_2 envoie des données au processeur P_1.) Après redistribution, le déséquilibre de P_2 doit être égal à zéro. Ainsi, $\delta_2 + \mathcal{R}_1 - \mathcal{R}_2 = 0$. Donc, comme $\mathcal{R}_1$ est connu, la valeur de $\mathcal{R}_2$ est aussi connue. En utilisant une récurrence directe, nous obtenons alors que, pour toute valeur de $j \in [2,n]$, $\mathcal{R}_j = \delta_j + \mathcal{R}_{j-1}$, et donc, que $\mathcal{R}_j$ est connu. Comme $\sum_{i=1}^{n} \delta_i = 0$, nous pouvons vérifier que $\delta_1 + \mathcal{R}_n - \mathcal{R}_1 = 0$. ■

Lemme 4.11. *Soit $\mathcal{R}$ une solution optimale rationnelle du problème de redistribution : pour tout j dans $[1,n]$, $\mathcal{R}_j$ représente le nombre de données que le processeur P_j envoie au processeur P_{j+1}. Il existe alors une solution entière optimale $\mathcal{E}$ telle que $|\mathcal{E}_1 - \mathcal{R}_1| < 1$.*

Démonstration. Nous prouvons le lemme 4.11 par contradiction. Nous supposons qu'aucune solution entière optimale $\mathcal{E}$ ne satisfait $|\mathcal{E}_1 - \mathcal{R}_1| < 1$. Prenons donc une solution entière optimale $\mathcal{E}$ telle que $|\mathcal{E}_1 - \mathcal{R}_1| \geqslant 1$. Soit $\mathcal{R}_1 = \mathcal{E}_1 + z + \epsilon$, où $z \in \mathbb{Z}$ et $\epsilon \in]-1;1[$ tels que $\mathcal{E}_1 + z \in [\mathcal{E}_1; \mathcal{R}_1]$. Donc

$$\mathcal{E}_1 \leqslant \mathcal{E}_1 + z \leqslant \mathcal{E}_1 + z + \epsilon \qquad \text{ou} \qquad \mathcal{E}_1 \geqslant \mathcal{E}_1 + z \geqslant \mathcal{E}_1 + z + \epsilon. \tag{4.9}$$

Ainsi, en utilisant la construction utilisée dans la preuve du lemme 4.10, nous avons :

$$\forall i \in [1,n], \mathcal{E}_i \leqslant \mathcal{E}_i + z \leqslant \mathcal{E}_i + z + \epsilon \qquad \text{ou} \qquad \forall i \in [1,n], \mathcal{E}_i \geqslant \mathcal{E}_i + z \geqslant \mathcal{E}_i + z + \epsilon. \tag{4.10}$$

Soit $\mathcal{F}$ une nouvelle solution entière de notre problème définie par : $\mathcal{F}_i = \mathcal{E}_i + z, \forall i \in [1,n]$. Alors, $|\mathcal{F}_1 - \mathcal{R}_1| = |(\mathcal{E}_1 + z) - (\mathcal{E}_1 + z + \epsilon)| = |\epsilon| < 1$. Si nous prouvons que $\mathcal{F}$ est une solution entière optimale, nous aurons atteint la contradiction voulue.

Considérons une valeur i dans $[1,n]$. Nous avons deux cas à traiter pour le processeur P_i (sous la redistribution $\mathcal{F}_i$) :

1. $\mathcal{F}_{i-1} \cdot \mathcal{F}_i \geqslant 0$: sous $\mathcal{F}_i$, soit le processeur P_i communique seulement des données avec l'un de ses voisins, soit il envoie ses données à l'un d'eux et reçoit des données de l'autre.
   Sans perte de généralité, supposons que $\mathcal{F}_{i-1} \geqslant 0$ et $\mathcal{F}_i \geqslant 0$. Nous devons alors montrer que
   $$\max\{\mathcal{F}_{i-1}c_{i-1,i}, \mathcal{F}_i c_{i,i+1}\} \leqslant \tau_{\text{int}},$$
   où τ_{int} est la durée de la solution entière optimale. Cependant, $\mathcal{F}_{i-1}c_{i-1,i} = (\mathcal{E}_{i-1} + z)c_{i-1,i}$. Si $\mathcal{E}_{i-1} + z$ est nul, $\mathcal{F}_{i-1}c_{i-1,i} = 0 \leqslant \tau_{\text{int}}$. Sinon, $\mathcal{E}_{i-1} + z$ est non nul. Comme $\mathcal{E}_{i-1} + z$ est par définition un entier, et comme $|\epsilon| < 1$, $\mathcal{E}_{i-1} + z$ et $\mathcal{E}_{i-1} + z + \epsilon$ ont le même signe, donc sont (strictement) positifs, et donc dans les deux redistributions les données sont envoyées du processeur P_{i-1} au processeur P_i.
   – Si $\epsilon > 0$, alors
   $$(\mathcal{E}_{i-1} + z)c_{i-1,i} < (\mathcal{E}_{i-1} + z + \epsilon)c_{i-1,i} = \mathcal{R}_{i-1}c_{i-1,i} \leqslant \tau_{\text{rat}} \leqslant \tau_{\text{int}},$$
   comme $\mathcal{R}$ est par définition une solution optimale rationnelle, et comme les solutions optimales rationnelles ne sont pas plus mauvaises que les solutions entières optimales.
   – Si $\epsilon < 0$, alors
   $$(\mathcal{E}_{i-1} + z + \epsilon)c_{i-1,i} < (\mathcal{E}_{i-1} + z)c_{i-1,i} < \mathcal{E}_{i-1}c_{i-1,i} \leqslant \tau_{\text{int}},$$
   du fait de l'équation 4.10, et comme $\mathcal{E}$ est par définition une solution entière optimale.
2. $\mathcal{F}_{i-1} \cdot \mathcal{F}_i < 0$: soit P_i reçoit des données de ses deux voisins, soit P_i leur envoie des données à tous les deux. Sans perte de généralité, supposons que P_i envoie des données aux deux.
   Nous devons alors montrer que
   $$-\mathcal{F}_{i-1}c_{i,i-1} + \mathcal{F}_i c_{i,i+1} \leqslant \tau_{\text{int}}. \tag{4.11}$$
   Cependant, $-\mathcal{F}_{i-1}c_{i,i-1} + \mathcal{F}_i c_{i,i+1} = -(\mathcal{E}_{i-1} + z)c_{i,i-1} + (\mathcal{E}_i + z)c_{i,i+1}$. Comme $\mathcal{E}_{i-1} + z$ est par définition un entier et comme $|\epsilon| < 1$, $\mathcal{E}_{i-1} + z$ et $\mathcal{E}_{i-1} + z + \epsilon$ ont le même signe, donc sont (strictement) négatifs, et donc des données sont envoyées du processeur P_i au processeur P_{i-1} dans les deux redistributions. De façon similaire, des données sont envoyées du processeur P_i au processeur P_{i+1} dans les deux redistributions.
   Comme $\mathcal{R}$ est par définition une solution optimale rationnelle, et comme les solutions optimales rationnelles ne sont pas plus mauvaises que les solutions entières optimales, alors :
   $$-(\mathcal{E}_{i-1} + z + \epsilon)c_{i,i-1} + (\mathcal{E}_i + z + \epsilon)c_{i,i+1} = -\mathcal{R}_{i-1}c_{i,i-1} + \mathcal{R}_i c_{i,i+1} \leqslant \tau_{\text{rat}} \leqslant \tau_{\text{int}}.$$
   Donc, si $\epsilon(c_{i,i+1} - c_{i,i-1}) \geqslant 0$, l'équation 4.11 est vérifiée. Sinon, $\epsilon(c_{i,i+1} - c_{i,i-1}) < 0$ et nous avons deux cas à considérer, dépendant de la redistribution $\mathcal{E}$:

– $\mathcal{E}_{i-1} \cdot \mathcal{E}_i < 0$: alors $\mathcal{E}_{i-1} < 0$ et $\mathcal{E}_i > 0$. En effet, quelle que soit la redistribution $\mathcal{S}$, nous avons toujours $\delta_i + \mathcal{S}_{i-1} - \mathcal{S}_i = 0$. Comme nous avons supposé que $\mathcal{F}_{i-1} < 0$ et $\mathcal{F}_i > 0$ alors $\delta_i > 0$, lequel interdit d'avoir $\mathcal{E}_{i-1} > 0$ et $\mathcal{E}_i < 0$.
Comme $\mathcal{E}$ est une solution entière optimale, nous avons :

$$-\mathcal{E}_{i-1}c_{i,i-1} + \mathcal{E}_i c_{i,i+1} \leqslant \tau_{\text{int}}.$$

L'équation 4.10 implique que z et ϵ sont tous les deux de même signe. Donc, $z(c_{i,i+1} - c_{i,i-1}) < 0$. D'où

$$-\mathcal{F}_{i-1}c_{i,i-1}+\mathcal{F}_i c_{i,i+1} = -(\mathcal{E}_{i-1}+z)c_{i,i-1}+(\mathcal{E}_i+z)c_{i,i+1} < -\mathcal{E}_{i-1}c_{i,i-1}+\mathcal{E}_i c_{i,i+1} \leqslant \tau_{\text{int}}.$$

– $\mathcal{E}_{i-1}\cdot\mathcal{E}_i \geqslant 0$. Sans perte de généralité, supposons que $\epsilon > 0$. Alors, $c_{i,i+1}-c_{i,i-1} < 0$. Du fait de l'équation 4.10, comme $\epsilon > 0$ et comme $(\mathcal{E}_{i-1} + z) < 0$, $\mathcal{E}_{i-1} < 0$, et donc $\mathcal{E}_i \leqslant 0$.

$$\begin{aligned} -(\mathcal{E}_{i-1} + z)c_{i,i-1} + (\mathcal{E}_i + z)c_{i,i+1} &= -\mathcal{E}_{i-1}c_{i,i-1} + \mathcal{E}_i c_{i,i+1} + z(c_{i,i+1} - c_{i,i-1}) \\ &< -\mathcal{E}_{i-1}c_{i,i-1} + \mathcal{E}_i c_{i,i+1}, \end{aligned}$$

comme $c_{i,i+1} - c_{i,i-1} < 0$. Cependant, $\mathcal{E}_i \leqslant 0$, donc

$$-(\mathcal{E}_{i-1} + z)c_{i,i-1} + (\mathcal{E}_i + z)c_{i,i+1} < -\mathcal{E}_{i-1}c_{i,i-1} \leqslant \tau_{\text{int}}$$

comme $\mathcal{E}$ est une solution entière optimale.

■

4.7 Cas général

4.7.1 Borne inférieure du temps d'exécution

Nous obtenons la borne inférieure du temps de redistribution suivante :

Lemme 4.12. *Soit τ le temps optimal de redistribution. Alors :*

$$\tau \geqslant \max \left\{ \begin{array}{l} \max\limits_{1\leqslant k\leqslant n,\ \delta_k>0} \delta_k \min\{c_{k,k-1}, c_{k,k+1}\}, \\ \max\limits_{1\leqslant k\leqslant n,\ \delta_k<0} -\delta_k \min\{c_{k-1,k}, c_{k+1,k}\}, \\ \max\limits_{\substack{1\leqslant k\leqslant n,\\ 1\leqslant l\leqslant n-2,\\ \delta_{k,k+l}>0}} \min\limits_{0\leqslant i\leqslant \delta_{k,k+l}} \max\{i\cdot c_{k,k-1}, (\delta_{k,k+l} - i)\cdot c_{k+l,k+l+1}\} \\ \max\limits_{\substack{1\leqslant k\leqslant n,\\ 1\leqslant l\leqslant n-2,\\ \delta_{k,k+l}<0}} \min\limits_{0\leqslant i\leqslant -\delta_{k,k+l}} \max\{i\cdot c_{k-1,k}, (-\delta_{k,k+l} - i)\cdot c_{k+l+1,k+l}\} \end{array} \right\} \tag{4.12}$$

Démonstration. Considérons un processeur P_i avec un déséquilibre positif ($\delta_i > 0$). Même si le processeur P_i peut envoyer des données à ses deux voisins, du fait du modèle 1-port, il ne peut leur envoyer des données simultanément. La meilleure solution pour qu'un processeur P_i envoie δ_i données est de les envoyer par le plus rapide de ses liens sortants. Cela demande alors au processeur P_i au moins un temps $\delta_i \times \min\{c_{i,i-1}, c_{i,i+1}\}$ pour envoyer δ_i données, quelle que soit la destination de ces données. Nous avons le résultat symétrique pour le cas $\delta_i < 0$. Par conséquent, les deux premières équations du système 4.12.

Considérons maintenant une tranche non triviale de processeurs consécutifs $C_{k,l}$. Par « non triviale », nous entendons que la tranche n'est pas réduite à un seul processeur (nous avons déjà traité ce cas) et qu'elle ne contient pas non plus l'ensemble des processeurs. Supposons que $\delta_{k,l} > 0$. Ainsi, dans n'importe quel schéma de redistribution, au moins $\delta_{k,l}$ données doivent être envoyées par $C_{k,l}$. Comme une tranche n'est pas réduite à un seul processeur, les deux processeurs aux extrémités de la tranche, P_k et P_l, peuvent envoyer simultanément des données à leurs voisins à l'extérieur de la tranche, P_{k-1} et P_{l+1} respectivement. Ainsi, pendant la redistribution, le processeur P_k envoie un certain nombre $i \in [0, \delta_{k,l}]$ de données au processeur P_{k-1}, dès que P_l envoie le reste des données à P_{l+1}, ce qui prend un temps $\max\{i \cdot c_{k,k-1}, (\delta_{k,l} - i) \cdot c_{l,l+1}\}$. Nous choisissons alors pour i une valeur qui minimise ce temps. Nous avons le résultat symétrique pour le cas $\delta_{k,l} < 0$. D'où les deux dernières équations du système 4.12. ■

4.7.2 Une approche heuristique

Nous ne savons pas si la borne donnée par le lemme 4.12 peut toujours être atteinte, mais nous n'avons pas de contre-exemple prouvant qu'elle ne l'est pas.

Lorsque la solution trouvée par le système 4.7 ne satisfait pas l'hypothèse de « redistribution légère », nous avons la possibilité de modifier le système pour la forcer : nous obtenons le système 4.13 qui trouve une solution satisfaisant l'hypothèse de « redistribution légère » s'il en existe une. Mais il n'y a pas de raison *a priori* pour que la solution du système 4.13 soit optimale.

MINIMISER τ, AVEC LES CONTRAINTES SUIVANTES :

$$\begin{cases} \mathcal{S}_{i,i+1} \geqslant 0 & 1 \leqslant i \leqslant n \\ \mathcal{S}_{i,i-1} \geqslant 0 & 1 \leqslant i \leqslant n \\ \mathcal{S}_{i,i+1} + \mathcal{S}_{i,i-1} - \mathcal{S}_{i+1,i} - \mathcal{S}_{i-1,i} = \delta_i & 1 \leqslant i \leqslant n \\ \mathcal{S}_{i,i+1} c_{i,i+1} + \mathcal{S}_{i,i-1} c_{i,i-1} \leqslant \tau & 1 \leqslant i \leqslant n \\ \mathcal{S}_{i+1,i} c_{i+1,i} + \mathcal{S}_{i-1,i} c_{i-1,i} \leqslant \tau & 1 \leqslant i \leqslant n \\ \mathcal{S}_{i,i+1} + \mathcal{S}_{i,i-1} \leqslant L_i & 1 \leqslant i \leqslant n \end{cases} \tag{4.13}$$

Pour conclure cette section, nous précisons que la conception d'un algorithme optimal dans le cas général reste ouvert. Étant donné la complexité de la borne inférieure, le problème semble très difficile à résoudre.

4.8 Simulations

Pour évaluer l'impact de la redistribution de données, nous avons utilisé le simulateur SIMGRID [54] pour modéliser une application itérative sur une plate-forme générée par Tiers [13, 26].

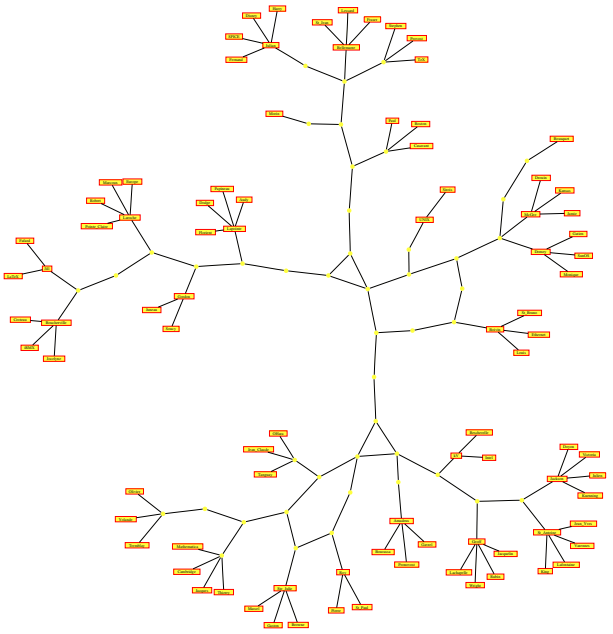


FIG. 4.11 – La plate-forme est composée de 90 nœuds, connectés par 192 liens de communication.

Nous utilisons la plate-forme représentée à la figure 4.11. Cette plate-forme est générée de la même façon que les plates-formes de la section 3.4.5.2. Les capacités des arêtes sont attribuées en utilisant la classification de Tiers (lien local LAN, lien LAN/MAN, lien MAN/WAN, ...). Pour chaque type de lien, nous utilisons les valeurs mesurées avec `pathchar` [27] entre plusieurs machines situées en France (Strasbourg, Lille, Grenoble et Orsay), aux États-Unis (Knoxville, San Diego et Argonne), et au Japon (Nagoya et Tokyo).

Nous avons sélectionné aléatoirement des processeurs dans notre plate-forme afin de construire un anneau. La vitesse de communication est donnée par le lien le plus lent sur une route allant d'un processeur à son successeur (ou prédécesseur) dans l'anneau. Les vitesses de calcul des processeurs sont tirées aléatoirement dans une liste de valeurs correspondant à des puissances de calcul (en MFlops et évaluées grâce à un jeu de tests de LINPACK [11]) d'une grande variété de machines (Pentium Pro 200 MHz, Pentium 2 350 MHz, Céléron 400 MHz, Athlon 1,4 GHz, Pentium 4 1,7 GHz, ...). Puis nous faisons varier ces vitesses pendant l'exécution de l'algorithme.

L'application exécute 100 itérations. À chaque itération, des données indépendantes sont mises-à-jour par les processeurs. Nous pouvons penser à une matrice

$m \times n$ de données dont les colonnes sont distribuées aux processeurs (nous utilisons $n = m = 1000$ dans nos simulations). Dans l'idéal, chaque processeur devrait avoir un nombre de colonnes proportionnel à sa puissance de calcul. C'est comme cela que la distribution des colonnes aux processeurs est initialisée.

De manière à motiver une redistribution des données, nous avons fait varier les vitesses de calcul des processeurs deux fois au cours de notre application itérative. La puissance de calcul de chaque processeur change une première fois aléatoirement entre les itérations 20 et 40, et une seconde fois entre les itérations 60 et 80 (cf. figure 4.12). Ces puissances de calcul sont enregistrées dans une trace de SIMGRID.

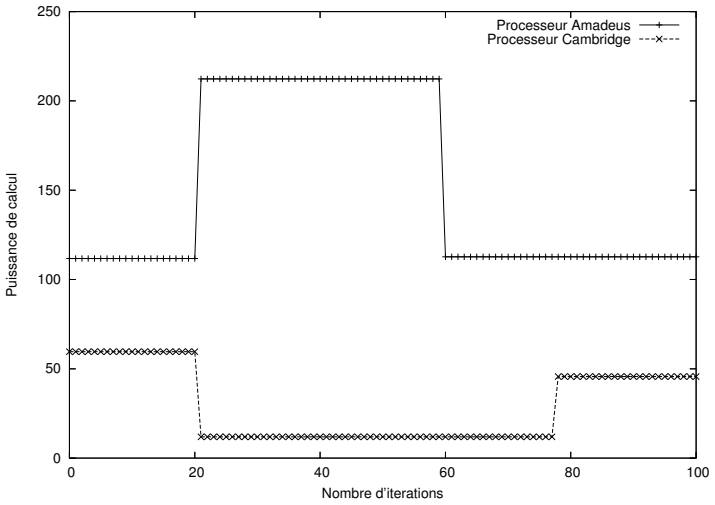


FIG. 4.12 – Puissance de calcul de 2 machines.

Pour les simulations, nous utilisons l'algorithme hétérogène bidirectionnel de la redistribution légère, et nous testons cinq schémas de redistribution différents sur 100 itérations. Les phases de redistributions ont eu lieu de la manière suivante :

- pas de redistribution ;
- une seule redistribution à la 50^{e} itération ;
- 4 redistributions, aux 20^{e}, 40^{e}, 60^{e} et 80^{e} itérations ;
- 9 redistributions (une toutes les 10 itérations) ;
- 19 redistributions (une toutes les 5 itérations).

À chaque itération, deux schémas peuvent avoir lieu :

- une phase de calcul : chaque processeur exécute les données qu'il possède en local.
- une phase de calcul et une phase de redistribution, donc une phase de communication au moment de la redistribution.

Nous avons pris en compte, afin de pouvoir comparer nos différentes phases de redistribution, le ratio (puissance de calcul)/(vitesse de communication). Plus ce paramètre décroît, plus les itérations prennent de temps et plus la redistribution de données devient nécessaire.

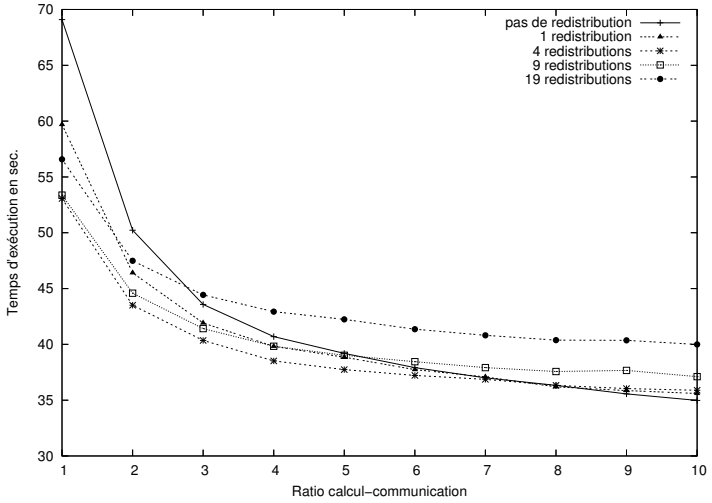


FIG. 4.13 – Temps d'exécution en fonction du quotient calcul-communication, pour un anneau de 8 processeurs.

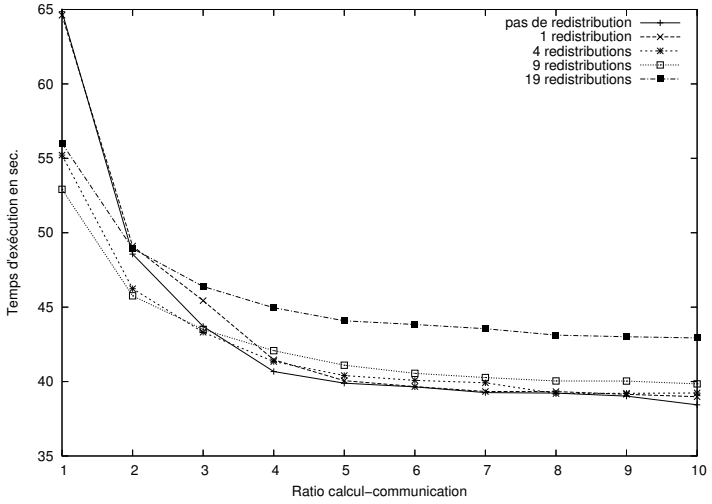


FIG. 4.14 – Temps d'exécution en fonction du quotient calcul-communication, pour un anneau de 32 processeurs.

Les figures 4.13 et 4.14 représentent le temps d'exécution des différents schémas de redistribution pour des tailles d'anneaux différentes : 8 processeurs et 32 processeurs.

Comme prévu, lorsque la puissance de calcul est élevée (ratio tendant vers 10 dans les figures), la meilleure stratégie est de n'employer aucune redistribution, car leur coût est prohibitif. Réciproquement, plus la puissance de calcul est faible (ratio tendant vers 1 dans les figures), plus les redistributions de données sont nécessaires, mais pas en trop grand nombre !

4.9 Conclusion

Nous nous sommes intéressée au problème de redistribution de données sur un anneau de processeurs. En ce qui concerne les anneaux homogènes (unidirectionnels ou bidirectionnels), le problème a été complètement résolu : nous avons des algorithmes optimaux ainsi que les preuves de leur optimalité. L'algorithme bidirectionnel s'est avéré complexe et a nécessité une longue preuve.

En ce qui concerne les anneaux hétérogènes, il reste un problème ouvert. Le cas unidirectionnel a été assez simple à résoudre ; par contre, le cas bidirectionnel n'a pas été totalement résolu : nous avons obtenu une solution optimale pour la redistribution légère. Cependant, la complexité de la borne dans le cas général montre qu'obtenir un algorithme optimal est une tâche ardue, si un tel algorithme existe.

Tous nos algorithmes ont été implémentés et intensivement testés. Nous avons reporté quelques résultats de simulation pour le cas le plus difficile, c'est-à-dire un anneau hétérogène bidirectionnel. Comme il était supposé, la redistribution de données peut être très efficace dans certains cas comme inutile dans d'autres.

Jusqu'à présent nous avons travaillé sur des plates-formes hétérogènes en prenant en compte la semi-dynamicité de ces plates-formes. Il serait donc intéressant de nous orienter vers la dynamicité complète des plates-formes. Ce cas fait l'objet d'une discussion de plusieurs méthodes dans le chapitre suivant.

Chapitre 5

Perspectives et conclusion

Une belle idée qui n'aboutit pas vaut mieux qu'une mauvaise qui voit le jour.
Pierre Dac

5.1 Variation de performances sur plates-formes hétérogènes

L'objectif principal de cette section est d'étudier un problème d'optimisation plus complexe, qui minimise T_{step} lorsque les caractéristiques de la plate-forme varient de façon significative pendant l'exécution de l'algorithme. Ces variations apparaissent par exemple avec l'arrivée d'un autre utilisateur sur la plate-forme, ou bien encore à cause de la maintenance système des nœuds, voire des pannes, etc. La solution courante (anneau virtuel, allocation de bande passante, distribution de données) peut ne plus être efficace avec les nouvelles caractéristiques de la plate-forme. Le problème est de décider *quand* et *comment* changer la solution courante.

Comme dans les chapitres précédents, nous allons nous intéresser aux algorithmes itératifs sur plates-formes hétérogènes. Notre contrainte géométrique d'organiser les processeurs en anneau virtuel reste bien sûr valable, et donc, nous avons comme précédemment, les notations suivantes :

- $T_{\text{step}}(i)$, le temps nécessaire au processeur P_i pour exécuter ses calculs en local et échanger ses données avec ses voisins dans l'anneau.
- T_{step}, la durée totale d'une itération, c'est-à-dire le maximum de $T_{\text{step}}(i)$ sur l'ensemble des processeurs P_i.

Comme nous l'avons vu au chapitre 3, la minimisation de T_{step} est déjà un problème difficile lorsque les caractéristiques de la plate-forme de calcul sont stables.

Il existe plusieurs *stratégies de transition* possibles pour mettre à jour la solution courante. Nous pouvons simplement garder la solution courante et chercher une

meilleure redistribution de données dans l'anneau. Si nous en trouvons une, nous redistribuons les données. Nous pouvons aussi construire un nouvel anneau. Par exemple il peut être nécessaire de remplacer un ou plusieurs processeurs qui sont devenus trop lents. Pour résumer, nous allons considérer les solutions suivantes :

Mises-à-jour locales– Nous pouvons essayer de remplacer le plus « mauvais » processeur, par exemple P_i, c'est-à-dire le processeur d'indice i tel que $T_{\text{step}}(i)$ est maximal. Nous insérons un nouveau processeur, par exemple P_j, dans l'anneau, à la place de P_i, et nous transférons les données locales de P_i à P_j. Notons que nous avons à calculer une nouvelle allocation des bandes passantes sur la totalité de l'anneau, parce que les chemins entre P_j et ses voisins peuvent interférer avec les autres chemins de l'anneau.

Redistribution de données– Nous pouvons calculer une nouvelle allocation de données sur l'anneau courant, ou à la fin de chaque mise-à-jour locale, ou seulement lorsque toutes les mises-à-jour locales ont été effectuées. Nous arrêtons la procédure de mise-à-jour dès qu'aucune amélioration ne peut être trouvée en remplaçant une ressource.

Changement global– Dans ce cas de figure, nous calculons une nouvelle solution (anneau et allocation de données) à partir de zéro, et nous faisons migrer les données de l'ancien anneau vers le nouvel anneau.

Pour toutes les approches précédentes, nous avons besoin d'une évaluation précise du coût encouru par tous les transferts de données requis pour migrer de la solution courante à la nouvelle. En prenant en compte le coût de cette redistribution, le gain atteint pour T_{step}, et le nombre d'itérations restantes, nous serons capables de sélectionner la meilleure alternative.

5.1.1 Stratégies de transition

Dans cette section, nous détaillons les différentes approches qui peuvent être utilisées pour faire face aux changements des caractéristiques de la plate-forme. Nous commençons par détailler le mécanisme de décision (*quand* modifier la solution courante) parce qu'il est basé sur l'analyse du coût pour toutes les stratégies de transition.

5.1.1.1 Évaluation d'une transition

Nous avons besoin de plusieurs notations afin d'expliquer comment nous calculons le coût d'un changement dans la solution courante. Nous caractérisons la solution courante de la façon suivante :

- nous avons un anneau $\mathcal{R} = \{P_1, \ldots, P_q\}$,
- nous avons des données allouées $\alpha_1, \cdots, \alpha_q$, où α_i représente les données attribuées à P_i,
- nous avons T_{step}, la durée d'une itération sur l'anneau $\mathcal{R}$.

S'il reste n_{step} itérations à exécuter, le temps total restant jusqu'à la fin de l'exécution de l'algorithme est estimé être $n_{\text{step}} \times T_{\text{step}}$.

Nous supposons maintenant que nous avons l'opportunité d'utiliser une nouvelle solution $\mathcal{R}'$, α', avec un temps T'_{step} par itération, qui serait calculée pour faire face aux changements des caractéristiques de la plate-forme. Faire la transition de la solution actuelle à la nouvelle, en considérant la redistribution des données ainsi que les itérations restantes, nécessite un temps total $T_{\text{redistrib}} + n_{\text{step}} \times T'_{\text{step}}$, où $T_{\text{redistrib}}$ est le temps nécessaire pour redistribuer les données $\alpha_1, \cdots, \alpha_q$ de la solution courante à l'allocation de données $\alpha'_1, \cdots, \alpha'_p$ du nouvel anneau. Nous pouvons voir que ce changement est valable seulement si

$$T_{\text{redistrib}} < n_{\text{step}} \times (T_{\text{step}} - T'_{\text{step}}).$$

Nous avons montré au chapitre précédent que dans de nombreuses situations, nous serons capables de calculer exactement $T_{\text{redistrib}}$, mais dans certains cas, nous ne pourrons pas calculer $T_{\text{redistrib}}$ et nous utiliserons alors une borne supérieure du temps nécessaire pour redistribuer les données, c'est-à-dire en supposant que toutes les communications impliquées dans la redistribution sont séquentielles. L'utilisation du pire cas comme scénario garantit que toutes les décisions de mises-à-jour seront profitables.

5.1.1.2 Mises-à-jour locales

Dans l'approche locale, nous mettons à jour la solution courante de manière incrémentale. Étant donné un anneau $\mathcal{R}$, nous sélectionnons un processeur P_i dont le $T_{\text{step}}(i)$ est maximal. Pour chaque processeur P_j n'étant pas inclus dans l'anneau, nous regardons s'il est profitable de remplacer P_i par P_j.

Il existe deux scénarios dans ce cas, un sans redistribution de données et un avec redistribution de données. Dans les deux cas, remplacer P_i par P_j nécessite de calculer les nouvelles allocations de bandes passantes pour l'anneau complet. Afin d'améliorer ce calcul, nous utilisons toujours KINSOL. Il y a cependant deux variantes :

- Dans le cas sans redistribution, nous gardons l'allocation actuelle, c'est-à-dire les valeurs des α_i, lorsque nous faisons appel à KINSOL. L'idée est de minimiser les mouvements de données. Nous payons seulement le coût de migration de α_i données de P_i à P_j, donc $T_{\text{redistrib}} = \frac{\alpha_i}{b_{i,j}}$, où $b_{i,j}$ est la bande passante du chemin $\mathcal{C}_{i,j}$ de P_i à P_j.
- Dans le cas avec redistribution, nous calculons les nouvelles valeurs α'_i lorsque nous faisons appel à KINSOL. L'idée est de minimiser T_{step}. Le prix supplémentaire à payer est une redistribution sur le nouvel anneau $\mathcal{R}'$. Nous utiliserons alors les algorithmes détaillés dans le chapitre 4 pour effectuer la redistribution.

Lorsque nous avons trouvé (et inséré) un nouveau processeur, nous pouvons recommencer le processus et essayer de remplacer un autre processeur tant que la valeur de T_{step} décroît. Si nous utilisons le scénario sans redistribution, il serait préférable de faire une redistribution après que plusieurs processeurs aient été remplacés.

5.1.1.3 Changement global

C'est une méthode brutale qui consiste à calculer une nouvelle solution (anneau et allocation des données) à partir de zéro. Lorsque cela est fait, il reste à migrer toutes les données de l'ancien anneau vers le nouveau. Rappelons que la nouvelle solution est construite de façon gloutonne, comme la première, donc que le premier processeur est bien défini dans les deux anneaux. De façon arbitraire, ce premier processeur donne l'origine de la redistribution. (Nous pourrions bien évidemment regarder les différentes possibilités qui s'offrent à nous en changeant l'origine de l'anneau et prendre celle qui nous donne le meilleur temps en considérant la redistribution de données et les itérations restantes.) De plus, nous connaissons la répartition de la charge de travail sur le nouvel anneau. Nous effectuons alors un calcul pour chaque processeur de l'ancien anneau (resp. du nouvel anneau) afin de déterminer à qui (resp. de qui) chaque processeur enverra (resp. recevra) des données. Une fois cette étape terminée, l'échange de données peut avoir lieu. Afin d'obtenir une première estimation du coût de ce transfert, nous utilisons le pire cas comme scénario : nous supposons que les transferts sont exécutés séquentiellement. Cependant, comme il vient d'être dit, il s'agit du pire cas et nous sommes amenée à penser que le coût du transfert de données sera moins élevé, notamment si certaines communications ont lieu en parallèle. Les communications ayant lieu en parallèle peuvent partager ou non certains liens de communication. Nous avons alors plusieurs possibilités en ce qui concerne l'estimation du coût de transfert des données :

1. Nous pourrions utiliser un algorithme calculant un maximum de chemins disjoints [36], c'est-à-dire trouver le maximum de couples de processeurs (P_j, N_k) (où P_j est le j^{e} processeur de l'ancien anneau et N_k le k^{e} processeur du nouvel anneau) devant échanger des données, tels que les chemins entre ces couples soient des chemins à arêtes disjointes. Mais un risque d'échec existe.
2. Nous pourrions aussi partager la bande passante. Dans ce cas de partage, plusieurs possibilités s'offrent à nous :
   - de façon gloutonne, réserver toute la bande passante disponible sur les chemins de communication entre chaque paire de processeurs de l'anneau ; les liens de communication seront alors saturés et ne seront plus utilisables pour d'autres chemins, ce qui peut avoir comme conséquence d'isoler certains processeurs ; nous avons donc encore un risque d'échec ;
   - réserver une fraction de la bande passante, équitable entre tous les chemins qui partagent le lien de communication. Et donc dans ce cas, nous permettons l'utilisation de tous les liens de communication sans saturation d'un lien en particulier.

   Ces deux premières méthodes ont cependant le désavantage de privilégier les premiers processeurs de l'anneau puisque nous construisons les chemins deux à deux en partant de la première paire de processeurs dans l'anneau. Afin de palier à ce « privilège », une méthode serait de :
   - résoudre un programme quadratique similaire au système quadratique 3.6 décrit dans la section 3.4.2 à l'aide de KINSOL, afin de répartir au mieux la bande

passante sur l'ensemble des chemins de communication.

3. Nous pourrions faire une simulation du transfert des données à l'aide de SIMGRID. La simulation des communications dans SIMGRID est basée sur les travaux de modélisation réaliste du partage de bande passante dans TCP, développés par H. Casanova et L. Marchal, et détaillés dans [54].

Nous obtenons donc, quelle que soit la méthode d'estimation employée (séquentielle, résolution quadratique avec KINSOL, simulation à l'aide de SIMGRID) une estimation de $T_{\text{redistrib}}$.

5.1.2 Perspectives

Il serait intéressant de simuler ces différentes approches afin de les comparer entre elles. Une possibilité serait d'écrire ces différentes méthodes et, suivant le cas dans lequel nous nous trouvons au moment de la prise de décision (par exemple, une variation de performances importante s'est produite), choisir la meilleure en fonction de son coût et du nombre d'itérations restantes.

5.2 Conclusion

Les différents chapitres de cette thèse ont permis de mesurer les difficultés algorithmiques liées à l'introduction de l'hétérogénéité de la plate-forme dans les modèles classiques. Les problèmes d'équilibrage de charge et de redistribution de données étaient déjà difficiles en environnement homogène, il n'y a aucune raison qu'ils se simplifient en environnement hétérogène. Ils ont nécessité la mise en œuvre d'algorithmes sophistiqués.

Nos principales contributions sont les suivantes :

Placement et équilibrage de charge : nous nous sommes tout d'abord intéressée au problème de l'équilibrage de charge en environnement homogène puis en environnement hétérogène en incluant le problème de contention des liens de communication. Nous avons montré en section 3.4.3 que le problème d'optimisation associé SHAREDRING était NP-complet, et nous avons proposé en section 3.4.4 une heuristique en temps polynomial pour résoudre ce problème. La section 3.5 montre que le partage des liens de communication a un impact important sur les stratégies d'équilibrage de charge.

Redistribution de données : nous nous sommes intéressée au problème de la redistribution de données sur un anneau de processeurs. Nous avons proposé en section 4.5.2 un algorithme de redistribution optimal pour anneaux homogènes bidirectionnels. Nous n'avons pas trouvé d'algorithme optimal pour le cas d'un anneau hétérogène bidirectionnel. Cependant, si nous supposons que chaque processeur possède initialement plus de données que ce qu'il doit envoyer pendant l'exécution de l'algorithme, alors nous réussissons à obtenir une solution

optimale. Cette solution est décrite en section 4.6.1. De plus, nous avons montré en section 4.8 l'impact des redistributions qui peuvent s'avérer très efficaces dans de nombreux cas.

La première étape à réaliser, selon nous, pour concevoir une application destinée à un environnement fortement hétérogène, semble être de commencer par ignorer les instabilités de la plate-forme et de développer des méthodes statiques dont nous pourrons essayer de garantir l'efficacité. Ensuite, en fonction de cette première étape, il sera possible d'établir des algorithmes en mesure de s'adapter aux paramètres dynamiques, tels que les variations de charge, tout en exhibant les mêmes bonnes propriétés d'optimalité que pour l'algorithme statique. Il existe bien sûr des cas où l'état de la plate-forme est totalement imprévisible. Il devient alors impossible de garantir les performances d'un algorithme et encore moins son optimalité. Il est donc important qu'il soit efficace dès que l'état de la plate-forme est stabilisé et que l'on soit en mesure de quantifier son efficacité, d'où la préférence de l'utilisation de modèles simples. Tous les algorithmes et heuristiques présentés dans cette thèse s'appuient justement sur des modèles simples, mais suffisamment réalistes pour permettre une modélisation raisonnablement précise du comportement des applications élaborées.

Annexe A

Bibliographie

[1] J. Bahi, S. Contassot-Vivier et R. Couturier. « Asynchronism for iterative algorithms in global computing environment ». Dans *16th Int. Symposium on High Performance Computing Systems and Applications* (Moncton, Canada, 2002), IEEE computer society press, pp. 90–97. p. 3

[2] J. Bahi, R. Couturier et F. Vernier. « Synchronous Distributed Load Balancing on Dynamic Networks ». *Journal of Parallel and Distributed Computing* **65** (juin 2005), 1397–1405. p. 11

[3] A. Bartoli, N. Nadal et R. Horaud. « Des séquences vidéo aux panoramas de mouvement ». Dans *COmpression et REpresentation des Signaux Audiovisuels* (2003). p. 3

[4] O. Beaumont, V. Boudet, A. Petitet, F. Rastello et Y. Robert. « A proposal for a heterogeneous cluster ScaLAPACK (dense linear solvers) ». *IEEE Trans. Computers* **50**, numéro 10 (2001), 1052–1070. p. 15

[5] M. Berkelaar. « LP_SOLVE ». `http://www.cs.sunysb.edu/~algorith/implement/lpsolve/implement.shtml`. p. 32

[6] F. Berman. « High-performance schedulers ». Dans *The Grid : Blueprint for a New Computing Infrastructure* (1999), I. Foster et C. Kesselman, éditeurs, Morgan-Kaufmann, pp. 279–309. p. 12

[7] D. Bertsekas et R. Gallager. *Data Networks*. Prentice Hall, 1987. p. 6, p. 43

[8] A. Bevilacqua. « A dynamic load balancing method on a heterogeneous cluster of workstations ». *Informatica* **23**, numéro 1 (1999), 49–56. p. 19

[9] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker et R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997. p. 18

[10] A. Bourgeade et B. Nkonga. « Dynamic load balancing computation of pulses propagating in a nonlinear medium ». *The Journal of Supercomputing* **28**, numéro 3 (2004), 279–294. p. 19

[11] R. P. Brent. « The LINPACK Benchmark on the AP1000 : Preliminary Report ». Dans *CAP Workshop 91* (1991), Australian National University. Website http://www.netlib.org/linpack/. p. 44, p. 81

[12] L. Brunie, A. Flory et H. Kosch. « New static scheduling and elastic load balancing methods for parallel query processing ». Dans *Basque International Workshop on Information Technology BIWIT* (1995), IEEE Computer Society Press. p. 19

[13] K. L. Calvert, M. B. Doar et E. W. Zegura. « Modeling Internet Topology ». *IEEE Communications Magazine* **35**, numéro 6 (juin 1997), 160–163. Available at http://citeseer.nj.nec.com/calvert97modeling.html. p. 8, p. 43, p. 81

[14] C.H.Hsu, Y. Chung, D. Yang et C. Dow. « A generalized processor mapping technique for array redistribution ». *IEEE Trans. Parallel Distributed Systems* **12**, numéro 7 (2001), 743–757. p. 18

[15] Y. Chow et W. Kohler. « Models for dynamic load balancing in a heterogeneous multiple processor system ». *IEEE Trans. Comput.* **28**, numéro 5 (1979), 354–361. p. 18

[16] M. Cierniak, M. Zaki et W. Li. « Compile-time scheduling algorithms for heterogeneous network of workstations ». *The Computer Journal* **40**, numéro 6 (1997), 356–372. p. 12, p. 13, p. 14

[17] M. Cierniak, M. Zaki et W. Li. « Customized dynamic load balancing for a network of workstations ». *Journal of Parallel and Distributed Computing* **43** (1997), 156–162. p. 12, p. 12

[18] S. Contassot-Vivier et J. Bahi. « Convergence dans les systèmes booléens asynchrones et applications aux réseaux de Hopfield ». *RSRCP (Réseaux et Systèmes Répartis, Calculateurs Parallèles), Numéro thématique : Algorithmes itératifs parallèles et distribués* **13**, numéro 1 (2001), 107–124. p. 2

[19] T. H. Cormen, C. E. Leiserson et R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990. p. 29, p. 42

[20] I. Craw. « Class notes, Linear Optimisation and Numerical Analysis, Mathematical Sciences, University of Aberdeen ». URL : http://www.maths.abdn.ac.uk/~igc/tch/mx3503/notes/node96.html. p. 29

[21] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian et T. v. Eicken. « LogP : Towards a realistic model of parallel computation ». Dans *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (1993), ACM Press, pp. 1–12. p. 13, p. 13

[22] G. Cybenko. « Dynamic load balancing for distributed memory multiprocessors ». *J. Parallel Distrib. Comput.* **7**, numéro 2 (1989), 279–301. p. 11

[23] F. Daïm, R. Eymard, D. Hilhorst, M. Mainguy et R.Masson. « Un algorithme de gradient conjugué préconditionné pour le couplage de codes géomécanique et réservoir ». *Oil and Gas Science and Technology* **57**, numéro 5 (2002), 515–523. p. 2

[24] E. Deelman et B. Szymanski. « Dynamic load balancing in parallel discrete event simulation for spatially explicit problems ». Dans *PADS'98, 12th Workshop on Parallel and Distributed Simulation* (1998), IEEE Computer Society Press, pp. 46–53. p. 18, p. 18

[25] F. Desprez, J. Dongarra, A. Petitet, C. Randriamaro et Y. Robert. « Scheduling block-cyclic array redistribution ». *IEEE Trans. Parallel Distributed Systems* **9**, numéro 2 (1998), 192–205. p. 18

[26] M. Doar. « A Better Model for Generating Test Networks ». Dans *Proceedings of Globecom '96* (novembre 1996). Available at http://citeseer.nj.nec.com/doar96better.html. p. 8, p. 43, p. 81

[27] A. B. Downey. « Using Pathchar to Estimate Internet Link Characteristics ». Dans *Measurement and Modeling of Computer Systems* (1999), pp. 222–223. Available at http ://citeseer.nj.nec.com/downey99using.html. p. 44, p. 81

[28] C. Durand. « Détection itérative non cohérente d'une modulation codée à bits entrelacés ». Rapport de thèse de doctorat, ENST, décembre 2000. p. 4

[29] P. Feautrier. « Parametric Integer Programming ». *RAIRO Recherche Opérationnelle* **22** (septembre 1988), 243–268. Software available at http://www.prism.uvsq.fr/~cedb/bastools/piplib.html. p. 32

[30] P. Feautrier et N. Tawbi. « Résolution de Systèmes d'Inéquations Linéaires ; mode d'emploi du logiciel PIP ». Rapport technique 90-2, Institut Blaise Pascal, Laboratoire MASI (Paris), janvier 1990. p. 32

[31] P. Feautrier et N. Tawbi. « Résolution de Systèmes d'Inéquations Linéaires ; mode d'emploi du logiciel PIP ». Research Report 90.2, IBP-MASI, janvier 1990. p. 6

[32] J. Garcia, E. Ayguadé et J. Labarta. « A framework for integrating data alignment, distribution, and redistribution in distributed memory multiprocessors ». *IEEE Trans. Parallel Distributed Systems* **12**, numéro 4 (2001), 416–431. p. 18, p. 19

[33] M. R. Garey et D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979. p. 28, p. 31, p. 40

[34] A. S. Grimshaw et J. B. Weissman. « A framework for partitioning parallel computations in heterogeneous environments ». *Concurrency : practice and experience* **7** (1995), 455–478. p. 16

[35] D. Grosu et A. T. Chronopoulos. « Noncooperative load balancing in distributed systems ». *Jouranl of Parallel and Distributed Computing* **65** (2005), 1022–1034. p. 17

[36] V. Guruswami, S. Khanna, R. Rajaraman, B. Shepherd et M. Yannakakis. « Near-optimal hardness results and approximation algorithms for edge-disjoint paths and related problems ». *Annual ACM Symposium on Theory of Computing* (1999), 19–28. p. 88

[37] M. Hamdi et C. Lee. « Dynamic load balancing of data parallel applications on a distributed network ». Dans *9th International Conference on Supercomputing ICS'95* (1995), ACM Press, pp. 170–179. p. 13, p. 18

[38] A. Heddaya et K. Park. « Mapping Parallel Iterative Algorithms onto Workstation Networks ». Dans *Prodceedings of 3rd Int.'1 Symp. on High Performance Computing* (August 2–5 1994). p. 16

[39] K. Helsgaun. « An effective implementation of the Lin-Kernighan traveling salesman heuristic ». *European Journal of Operational Research* **126**, numéro 1 (2000), 106–130. Software available at http://www.dat.ruc.dk/~keld/research/LKH/. p. 29, p. 33

[40] Y. Hu et R. Blake. « Load balancing for unstructured mesh applications ». *Parallel and Distributed Computing Practices* **2**, numéro 3 (1999), 117–148. p. 15

[41] S. Ichikawa et S. Yamashita. « Static load balancing of parallel PDE solver for distributed computing environment ». Dans *PDCS'2000, 13th Int'l Conf. Parallel and Distributed Computing Systems* (2000), ISCA Press, pp. 399–405. p. 17

[42] M. Irani, P. Anandan, J. Bergen, R. Kumar et S. Hsu. « Efficient representations of video sequences and their applications », 1996. p. 3

[43] A. Kalinov et A. Lastovetsky. « Heterogeneous distribution of computations while solving linear algebra problems on networks of heterogeneous computers ». Dans *HPCN Europe 1999* (1999), P. Sloot, M. Bubak, A. Hoekstra et B. Hertzberger, éditeurs, LNCS 1593, Springer Verlag, pp. 191–200. p. 15

[44] E. T. Kalns et L. M. Ni. « Processor mapping techniques towards efficient data redistribution ». *IEEE Trans. Parallel Distributed Systems* **6**, numéro 12 (1995), 1234–1247. p. 18

[45] H. Kameda et C. Kim. « Optimal static load balancing of multi-class jobs in a distributed computer systems ». *Proceedings of the 10th international conference on distributed computing systems* (1990), 562–569. p. 18

[46] H. Kameda, J. Li, C. Kim et Y. Zhang. « Optimal load balancing in distributed computer systems ». Dans *Springer Telecommunication Networks And Computer Systems Series archive* (1997), p. 251. p. 18

[47] G. Karypis et V. Kumar. « Parallel Multilevel k-way Partitioning Scheme for Irregular Graphs ». Research Report 96-036, University of Minnesota, 1996. Available at http://www.cs.umn.edu/~karypis. p. 19

[48] D. Katabi, M. Handley et C. Rohrs. « Congestion control for high bandwidth-delay product networks ». Dans *Proceedings of the ACM 2002 conference on applications, technologies, architectures, and protocols for computer communications (SIGCOMM)* (2002), ACM Press, pp. 89–102. p. 24

[49] J. Knoop et E. Mehofer. « Distribution assignment placement : effective optimization of redistribution costs ». *IEEE Trans. Parallel Distributed Systems* **13**, numéro 6 (2002), 628–647. p. 18, p. 19

[50] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. S. Jr. et M. E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994. p. 18

[51] U. Kremer. « NP-Completeness of dynamic remapping ». Dans *Proceedings of the Fourth Workshop on Compilers for Parallel Computers* (Delft, The Netherlands, 1993). Also available as Rice Technical Report CRPC-TR93330-S. p. 18

[52] Z. Lan, V. Taylor et G. Bryan. « Dynamic load balancing of SAMR applications on distributed systems ». Dans *Proceedings of the ACM/IEEE Symposium on Supercomputing (SC'01)* (2001), IEEE Computer Society Press. p. 14, p. 19

[53] C. Lee et M. Hamdi. « Parallel image processing applications on a network of workstations ». *Parallel Computing* **21** (1995), 137–160. p. 18

[54] A. Legrand, L. Marchal et H. Casanova. « Scheduling Distributed Applications : The SIMGRID Simulation Framework ». Dans *Proceedings of the Third IEEE International Symposium on Cluster Computing and the Grid (CCGrid'03)* (May 2003), p. 138. p. 4, p. 8, p. 81, p. 89

[55] S. Lin et B. W. Kernighan. « An effective heuristic algorithm for the traveling salesman problem ». *Operations Research* **21** (1973), 498–516. p. 29, p. 33

[56] S. Miguet et Y. Robert. « Elastic load balancing for image processing algorithms ». Dans *Parallel Computation* (1992), H. Zima, éditeur, LNCS 591, Springer Verlag, pp. 438–451. p. 19

[57] N. Park, V. Prasanna et C. Raghavendra. « A framework for integrating data alignment, distribution, and redistribution in distributed memory multiprocessors ». *IEEE Trans. Parallel Distributed Systems* **10**, numéro 12 (1999), 1217–1240. p. 18

[58] A. Pinar et C. Aykanat. « Fast optimal load balancing algorithms for 1D partitioning ». *Journal of Parallel and Distributed Computing* **64** (2004), 974–996. p. 16

[59] L. Prylli et B. Tourancheau. « Fast runtime block-cyclic data redistribution on multiprocessors ». *J. Parallel Distributed Computing* **45** (1997), 63–72. p. 18

[60] K. Salamatian. « Transmission multimédia fiable sur Internet ». Rapport de thèse de doctorat, LRI, Université Paris-Sud, France, décembre 1999. p. 3

[61] D. Sarrut et S. Miguet. « ARAMIS : a remote access medical imaging system ». Dans *ISCOPE'99, 3rd International Symposium on Computing in Object-Oriented Parallel Environments* (1999), volume 1732 des *Lecture Notes in Computer Science*, Springer. p. 19

[62] K. Schloegel, G. Karypis et V. Kumar. « Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes ». *Journal of Parallel and Distributed Computing* **47** (1997), 109–124. p. 19

[63] K. Schloegel, G. Karypis et V. Kumar. « A Unified Algorithm for Load-balancing Adaptive Scientific Simulations ». Dans *Proceedings of the ACM/IEEE Symposium on Supercomputing (SC'00)* (2000), IEEE Computer Society Press. p. 19

[64] H.-Y. Shum et R. Szeliski. « Systems and experiments paper : Construction of panoramic mosaics with global and local alignment ». *International Journal of Computer Vision* **36**, numéro 2 (2000), 101–130. p. 3

[65] A. Taylor et A. Hindmarsh. « User documentation for KINSOL, a nonlinear solver for sequential and parallel computers ». Rapport technique UCRL-ID-131185, Lawrence Livermore National Laboratory, juillet 1998. p. 6, p. 43

[66] R. Thakur, A. Choudhary et J. Ramanujam. « Efficient algorithms for array redistribution ». *IEEE Trans. Parallel and Distributed Systems* **7**, numéro 6 (1996), 587–594. p. 18

[67] H. Willebeek-LeMair et P. Reeves. « Strategies for Dynamic Load Balancing on Highly parallel Computers ». *IEEE Transactions on Parallel and Distributed Systems* **4**, numéro 9 (September 1993). p. 13

[68] M.-Y. Wu. « On runtime parallel scheduling for processor load balancing ». *IEEE Trans. Parallel and Distributed Systems* **8**, numéro 2 (1997), 173–186. p. 18

Annexe B

Liste des publications

Articles parus dans des journaux internationaux

[A] A. Legrand, H. Renard, Y. Robert et F. Vivien. « Mapping and load-balancing iterative computations on heterogeneous clusters with shared links ». *IEEE Trans. Parallel and Distributed Systems* **15**, numéro 6 (2004), 546–558. p. 6, p. 7

[B] H. Renard, Y. Robert et F. Vivien. « Data redistribution algorithms for heterogeneous processor rings ». *Journal of High Performance Computing Applications* (2006 (à paraître)). p. 9

Actes de conférences internationales avec comité de lecture

[C] H. Renard, Y. Robert et F. Vivien. « Static Load-Balancing Techniques for Iterative Computations on Heterogeneous Clusters (*Distinguished paper*) ». Dans *Euro-Par'03 : International Conference on Parallel and Distributed Computing* (Klagenfurt, Autriche, 26–29 août 2003), LNCS 2790, Springer Verlag, pp. 148–159. p. 6

[D] A. Legrand, H. Renard, Y. Robert et F. Vivien. « Load-Balancing Iterative Computations on Heterogeneous Clusters with Shared Comunication Links ». Dans *HeteroPar'03 : International Workshop on "Algorithms and Tools for Parallel Computing on Heterogeneous Clusters" on Heterogeneous Networks* (Czwestochowa, Pologne, 7–10 septembre 2003), LNCS 3019, Springer Verlag, pp. 930–937. p. 7

[E] A. Legrand, H. Renard, Y. Robert et F. Vivien. « Mapping and Load-Balancing Iterative Computations on Heterogeneous Clusters ». Dans *EuroPVM/MPI'03 : European PVM/MPI Users' Group conference* (Venezia, Italie, 29 septembre–02 octobre 2003), LNCS 2840, Springer Verlag, pp. 586–594. p. 7

[F] H. Renard, Y. Robert et F. Vivien. « Data redistribution algorithms for heterogeneous processor rings ». Dans *HiPC'04 : International Conference on High Performance Computing* (Bangalore, Inde, 19–22 décembre 2004), LNCS 3296, Springer Verlag, pp. 123–132. p. 9

Actes de conférences nationales avec comité de lecture

[G] A. Legrand, H. Renard, Y. Robert et F. Vivien. « Placement et équilibrage de charge sur plates-formes hétérogènes ». Dans *15e Rencontres Francophones du Parallélisme des Architectures et des Systèmes* (La Colle sur Loup, France, 15–17 octobre 2003). p. 7

[H] H. Renard, Y. Robert et F. Vivien. « Algorithmes de redistribution de données pour anneaux de processeurs hétérogènes ». Dans *16e Rencontres Francophones du Parallélisme des Architectures et des Systèmes* (Le Croisic, France, 6–8 avril 2005). p. 9

Rapports de recherche

[I] H. Renard, Y. Robert et F. Vivien. « Static load-balancing techniques for iterative computations on heterogeneous clusters ». Rapport de recherche RR-2003-12, LIP, ENS Lyon, France, février 2003. Également disponible comme rapport de recherche INRIA RR-4745.

[J] A. Legrand, H. Renard, Y. Robert et F. Vivien. « Load-balancing iterative computations in heterogeneous clusters with shared communication links ». Rapport de recherche RR-2003-23, LIP, ENS Lyon, France, avril 2003. Également disponible comme rapport de recherche INRIA RR-4800.

[K] H. Renard, Y. Robert et F. Vivien. « Data redistribution algorithms for heterogeneous processor rings ». Rapport de recherche RR-2004-28, LIP, ENS Lyon, France, mai 2004. Également disponible comme rapport de recherche INRIA RR-5207.

Résumé :

Dans cette thèse, nous nous sommes intéressée à la mise en œuvre d'algorithmes itératifs sur des grappes hétérogènes. Ces algorithmes fonctionnent avec un volume important de données (calcul de matrices, traitement d'images, etc.), qui sera réparti sur l'ensemble des processeurs. À chaque itération, des calculs indépendants sont effectués en parallèle et certaines communications ont lieu. Il n'existe pas de raison *a priori* de réduire le partitionnement des données à une unique dimension et de ne l'appliquer que sur un anneau de processeurs unidimensionnel. Cependant, un tel partitionnement est très naturel et nous montrerons que trouver l'optimal est déjà très difficile.

Après cette étude sur le placement et l'équilibrage de charge pour plates-formes hétérogènes, nous nous sommes intéressée à la redistribution de données sur ces mêmes plates-formes, lorsque que les caractéristiques de ces dernières changent. En ce qui concerne les anneaux de processeurs homogènes, nous avons totalement résolu le problème : nous avons obtenu des algorithmes optimaux et prouvé leur exactitude dans le cas homogène et dans le cas hétérogène. En ce qui concerne les anneaux hétérogènes, le cas unidirectionnel a été totalement résolu, alors que le cas bidirectionnel reste ouvert. Cependant, sous l'hypothèse de redistribution légère, nous sommes capable de résoudre le problème de manière optimale.

Mots-clés :

Algorithmes itératifs, équilibrage de charge, plates-formes hétérogènes, partage de liens de communication, algorithmes de redistribution de données, anneaux hétérogènes, complexité.

Abstract:

In this thesis, we study iterative algorithms onto heterogeneous platforms. These iterative algorithms operate on large data samples (recursive convolution, image processing algorithms, etc.). At each iteration, independent calculations are carried out in parallel, and some communications take place. Note that there is no reason *a priori* to restrict to a uni-dimensional partitioning of the data, and to map it onto a unidimensional ring of processors. But uni-dimensional partitionings are very natural for most applications, and, as will be shown in this thesis, the problem to find the optimal one is already very difficult.

After dealing with the problems of mapping and load-balancing onto heterogeneous platforms, we consider the problem of redistributing data onto these platforms, an operation induced by possible variations in the resource performances (CPU speed, communication bandwidth) or in the system/application requirements (completed tasks, new tasks, migrated tasks, etc.). For homogeneous rings the problem has been completely solved. Indeed, we have designed optimal algorithms, and provided formal proofs of correctness, both for unidirectional and bidirectional rings. For heterogeneous rings there remains further research to be conducted. The unidirectional case was easily solved, but the bidirectional case remains open. Still, we have derived an optimal solution for light redistributions, an important case in practice.

Keywords:

Iterative algorithms, load-balancing, heterogeneous platforms, shared communication links, data redistribution algorithms, heterogeneous rings, complexity.